16. 最初のタイムシェアリング・マシンの1つ，PDP-1は18ビットワードを4K持っていた．メモリ内に常時1プロセスを保存していた．スケジューラが別のプロセスを実行しようとする時は，メモリ内のプロセスはページング・ドラムに書き込まれ，そのドラムには1周あたり18ビットワードが4K記録できた．しかもドラムはワード0からだけでなく，任意のワードから読み書きすることができた．このドラムがどうして使用されることになったのかを考えよ．

17. あるコンピュータは各プロセスに対して4,096バイトページに分割された65,536バイトのアドレス空間を割り当てている．特定のプログラムが32,768バイトのテキストサイズ，16,386バイトのデータサイズ，そして15,870バイトのスタックサイズを持っているとしよう．このプログラムはアドレス空間に適合するか．ページサイズが512バイトならどうか．1つのページは2つの異なるセグメントを含まないことに注意せよ．

18. ページフォルトとページフォルトの間に実行された命令数は，プログラムに割り当てられたページフレームの数に正比例していることがわかっている．使用可能なメモリが倍になったら，ページフォルト間の平均間隔も倍になる．通常の命令は1マイクロ秒しかかからないが，ページフォルトが発生すると2001マイクロ秒かかってしまう．プログラムの実行に60秒かかり，その間に15,000のページフォルトが発生するとしよう．この時2倍のメモリ量を与えた場合の，実行時間を求めよ．

19. MINIXのメモリ管理手法では，chmemの様なプログラムがなぜ必要なのか説明せよ．

20. MINIXプロセスがSIGNALシステムコールを発行し，シグナルを捕えられるようにした時，呼び出される関数のアドレスはメモリマネージャではなく，プロセス自身のテーブルに保存されることになる．この様な技法を用いることによる利点をあげよ．

21. 親プロセスが待機状態になるまで待たずに，ゾンビ状態になったプロセスのメモリを即座に解放できるようMINIXを修正せよ．

22. 現在のところMINIXでは，EXECシステムコールが発行されると，メモリマネージャが新しいメモリイメージを保存するのに十分な大きさのホールが存在しているかを調べる．それだけの大きさのホールが存在しなければ，呼出しは拒絶される．より高度なアルゴリズムでは，現在使用中のメモリイメージが解放された時，十分な大きさのホールが存在するかを調べることができる．このアルゴリズムを実現せよ．

23. EXECシステムコール実行時に，MINIXはファイルシステムにセグメント全体を一度に読み取らせるようなトリックを行う．同じような方法でコアダンプを書き込むためのトリック技法を説明し，実現せよ．

24. MINIXを修正し，スワッピングを可能にせよ．

5章
ファイルシステム

オペレーティング・システムにおいて，最も頻繁に利用されるのはファイルシステムである．プログラムでは，ファイルの読み書きが行われ，またユーザーはファイルの存在とその属性を常に意識している．オペレーティング・システムが使いやすく，そして便利なものであるかどうかは，ファイルシステムのインターフェイス，構造，信頼性に大きく依存している．

本章では，ユーザー側から見たファイルシステムのさまざまな形態，その実現方法，不正使用に対するファイルの保護，MINIXファイルシステムの設計と実現方法などを解説する．また分散オペレーティング・システムにおいて見られる設計上の問題点に関しても解説していく．

5.1 ユーザーから見たファイルシステム

ユーザーの立場において，ファイルシステムで最も重要なものは，その形態である．すなわちファイルを構成しているものが何か，その命名法，保護法，許可されるファイル操作，などが最大の関心事である．ファイルシステムの詳細，例えば空き領域を管理するために連結されたリストやビットマップをどの様に使用しているか，また論理ブロック内のセクタ数など，細かいことはそれほど重要ではない．ただし，ファイルシステムの設計者にとっては，この様な詳細が非常に重要である．以降の節では，ユーザーインターフェイスに関するいくつかの事項を説明する．章の後半ではファイルシステムの構築法に関して考察する．

5.1.1 ファイルに関する基礎

コンピュータに情報を保存するための最適な方法としては，各プロセスに非常に大きなセグメントを多数，例えば2^{32}の長さを持つセグメントを2^{32}個与えることである．誰かが最初にコンピュータを使い始めた時，シェルと標準のセグメント(ここにはエディタ，コンパイラ，その他ユーティリティなどが含まれるだろう)のためのアドレス空間が与えられる．時間が経過するにつれ，ユーザーの入力したテキスト，プログラム，その他の情報はすべてシェルのアドレス空間内に蓄積されていく．あるセグメントにはディレクトリが格納されており，階層構造のファイル管理(UNIXの様な)を行っている．

このアドレス空間は，すべてのプロセスによって継承され，必要があればセグメントを追加し，不要になればアドレス空間はシェルに返される．これにより各情報のアクセスは，メモリ(仮想)に対する読み書きだけの問題となる．

MULTICSではこの問題に真剣に取り組んだが，実際にはこの様な情報の格納方法はまだ実現されていない．その理由の1つには，現在のアドレス空間が小さすぎ，すべてのコンピュータが仮想アドレスを持っているわけでもないということが挙げられる．また，プロセスがクラッシュした場合，そのアドレス空間が失われ，アドレス空間を長期間(例えば月や年単位で)保存できなくなる，というのも理由の1つである．それでも技術が進歩すれば，ユーザーのプロセスのアドレス空間にユーザーの情報をすべて保存しておくという概念は，再度検討されるだろう．

その様な環境が整うまで，大半のオペレーティング・システムでは，これとは違った方法を用いて情報を保存するだろう．それは，ユーザーが名前付けできるファイル(files)を用いるという方法である．ファイルに，プログラムやデータ，その他の必要なものを保存する．ファイルは，どのプロセスのアドレス空間の一部でもない．オペレーティング・システムは，ファイルの作成，削除，読み書きを行う特殊な操作(システムコール)を提供しており，またファイルを管理している．

図5.1は一般的な3つのファイルの構成を示したものである．最初の方法は，単なるバイト列である．UNIXのファイルは，この方法で構成されている．2番目の方法は，固定サイズ長の連続したレコードである．任意のレコードを読み書きできるが，ファイルの途中にレコードを挿入する，あるいは削除するなどは行えない．CP/Mは，この方法を採っている．

3番目の方法は，ツリー構造のディスクブロックである．個々のブロックがn個のキーを持つレコードを持っている．レコードはキーを用いて検索することができ，新しいレコードはツリー内の任意の場所に挿入することができる．すでに満杯となっているブロックに対し，新たにレコードが加えられた場合，ブロックは2つに分割され，両方ともアルファベット順に正しく並べられたツリーに加えられる．この方法はメインフレームにおいてよく採用されており，ISAM(Indexed Sequential Access Method)と呼ばれている．

オペレーティング・システムでは，デバイスに依存しない(device independence)ことが重要である．すなわちファイルまたは装置がどこに存在していようと，同じアクセスを行えるように

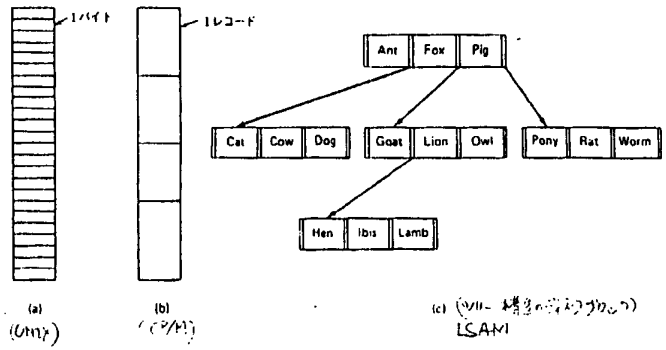

図5.1 3種類のファイル
(a)バイト列 (b)レコード列 (c)ツリー

することである．例えば，入力ファイルを読み取り，それをソートし，結果を出力ファイルを書き込むプログラムでは，フロッピーディスクやハードディスクなど，どのデバイスからでもそれを意識することなく入力が行え，またファイル，端末，あるいはプリンタなど，どのデバイスに対しても，デバイスを意識することなく出力が行えるということである．

あるオペレーティング・システムでは，他のオペレーティング・システムと比較して，装置に対する依存性が低いものもある．例えばUNIXではファイルシステム(例えばディスク)をファイルツリー内の任意の場所にマウントできるようになっており，すべてのファイルを，どの装置上にあるかを意識することなく，パス名を用いてアクセスできる．MS-DOSでは，ユーザーがどの装置上にファイルが存在しているかを明示的に指定しなければならない(ただし，デフォルトの装置となっているものだけは必要ない)．したがって，デフォルトの装置がドライブCであれば，ドライブA上にあるプログラムをドライブBに置かれた入出力ファイルを用いて実行するために，以下の様な入力を行わなければならない．

A : program < B : input > B : output

大半のオペレーティング・システムでは，多くの異なるファイル型を持っている．例えばUNIXには，通常のファイル，ディレクトリ，特殊ファイル(ブロックおよびキャラクタ)がある．通常のファイルはユーザーデータを含んでいる．ディレクトリにはファイルに記号名(すなわちASCII文字列)を付けるために必要な情報が格納されている．ブロックおよびキャラクタ型特殊ファイルは，それぞれディスク装置および端末装置などを表すために用いられる．これにより，例えばユーザーは以下の様な入力を行うことによって，ファイルabcを端末(キャラクタ型特殊ファイル/dev/tty)にコピーすることができる．

```
cp  abc  /dev/tty
```

UNIXのあるバージョンでは，名前付きパイプ(named pipe)と呼ばれる技法も用いられている．名前付きパイプは，2つのプロセスにおいてオープンし，そのプロセス同士で通信を行うことができる疑似ファイルである．ユーザーからすれば，これらのファイル型の違いは些細なものである(例えば端末に対してシークを行うことができないなど)．

大半のシステムでは，通常のファイルがさらにその用途に応じて異なる型に分類されている．個々の型は，ファイル名の最後に付けられるファイル拡張子(file extensions)によって識別できる．以下にその一例を示す．

FILE.PAS	Pascal ソースプログラム
FILE.FTN	FORTRAN ソースプログラム
FILE.BAS	BASIC ソースプログラム
FILE.OBJ	オブジェクトファイル
FILE.BIN	実行可能バイナリファイル
FILE.LIB	リンカの使用する.OBJ ファイルのライブラリ
FILE.TXT	フォーマッティング・プログラム用入力テキスト
FILE.DAT	データファイル

システムによっては拡張子を単なる慣習として使用しており，オペレーティング・システム自身はそれを特定の目的に使用していない．また，オペレーティング・システムが命名に関する厳しい制限を設けているシステムもある．例えば，.BIN で終了しない限りファイルを実行しないことになっている，という例もある．

TOPS-20システムでは，実行されるファイルの作成時間の管理まで行っている．そしてソースファイルを探し，バイナリが作成されてから後ソースが修正されたかを調べている．修正されていれば，自動的にソースをコンパイルし直す．UNIXシステムにおける make プログラムに該当するものが，シェルに内蔵されているのである．拡張子が必ず必要であるため，オペレーティング・システムは，どのバイナリプログラムがどのソースから作成されているのかを識別できる．

この様に，ファイルに対して強い型付けを行うと，ユーザーが，システム設計者の予測しなかったことを行おうとするたびに問題が発生してしまう．例えば出力結果として，.DAT(データファイル)型のファイルを生成するプログラムを考えてみる．ユーザーが.PAS ファイルを読み取り，それを変換し(例えば標準のインデント形式に変換する)，変換したファイルを出力へ書き込む場合，出力ファイルの型は.DAT となる．これを Pascal コンパイラを用いてコンパイルすると，拡張子が誤っているため，システムはコンパイルを行わない．FILE.DAT を FILE.PAS へコピーしようとしても，システムはコピーを拒否する(ユーザーの誤りを防ぐため)．

オペレーティング・システムの判断によってこの様な保護が行われることは，初心者にとってはありがたいものであるが，上級者にとってはうとましいものであろう．

ファイルに対して行える操作は，オペレーティング・システムによって異なっているが，バイト(あるいはレコード)の連続的な読み書きはどのシステムでも行える．ランダムアクセス(random access)は，効果的であると思われた場合に提供されている(通常のファイルに対しては使えるが，ラインプリンタの様な特殊ファイルでは不可)．あるオペレーティング・システムでは，READ システムコールを使用する際に読み書きするレコードの番号(あるいはキー)を指定し，ランダムアクセスが行われている．また，ファイル内における"現在のファイル位置"を設定するシステムコールが提供されており，それにより以降の READ または WRITE コールが新しい位置に対する操作を行う，というシステムもある(例えば UNIX など)．レコードを用いているファイルでは，レコードの挿入と削除の操作が行えるようになっている．

5.1.2 ディレクトリ

ファイルを記録しておくために，ファイルシステムでは通常ディレクトリ(directory)を提供しており，大半のシステムではこのディレクトリ自身もファイルの一種となっている．ディレクトリには通常いくつかのエントリがファイルごとに含まれている．これを示したのが図5.2である．最も簡単な方法では，システムがすべてのユーザーの全ファイルを持つ1つのディレクトリを管理している．多くのユーザーが存在する場合，それぞれのユーザーが同じファイル名を使おうとし(例えば mail や games など)，衝突や，混乱が生じ，システムが稼働できなくなる．この様なシステムは最も初歩的なマイクロコンピュータ用オペレーティング・システムにしか用いられていない．

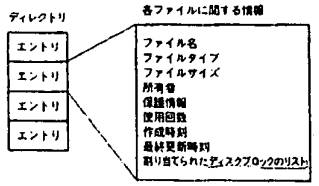

図5.2 ディレクトリ・エントリ
ディレクトリには1つのファイルに関するエントリを，複数ファイル分持たせることができる．エントリにはファイルに関する情報や，その様な情報を格納しておく他の構造体を指すポインタが含まれている．

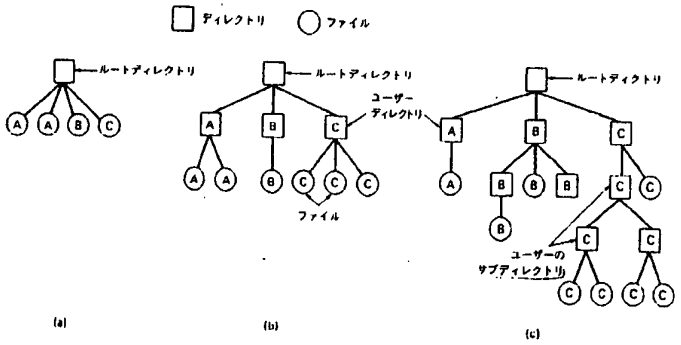

図5.3 3つのファイルシステム設計
(a)すべてのユーザーが1つのディレクトリを共有している
(b)1ユーザーあたり1ディレクトリ
(c)ツリー構造．図中の英字はファイルやディレクトリの所有者を示す

1つのディレクトリにすべてのファイルを持たせるという考えに改良を加え，1ユーザーあたり1つのディレクトリを持たせることもできる(図5.3(b)参照)．この技法はユーザー間でのファイル名の衝突を防ぐことにはなったが，多くのファイルを用いるユーザーにはあまり効果的ではなかった．論理的な方法で自分のファイルをグループ化しようとするユーザーも少なくなかった．例えば大学教授の場合，ある講義のために作成している資料のファイル，またそれをさらに本にまとめるファイル，別な講義に使用する学生用のプログラム，自分自身用に作成している高度なコンパイラ作成プログラム，承認済みの提案を格納しているファイル，メイル送受信用のファイル，議事録，書きかけの論文，ゲームなど，多種多様なファイルを持っており，これらを整理するための様々なファイルグループが必要となる．これらのファイルを論理的にまとめるための方法が必要である．

ここで必要となるのは，階層化(すなわちディレクトリ・ツリー)である．このアプローチにより，ユーザーはファイルが自然な方法でグループ分けできるように，必要なだけディレクトリを持つことができるようになる．これを示したのが図5.3(c)である．

ファイルシステムがディレクトリのツリーとして構成されると，ファイル名を指定する方法を決めておかなければならない．一般的には次の2つの方法が使用されている．最初の方法では，各ファイルに，ルートディレクトリからそのファイルまでのパスを表す絶対パス名(absolute path name)を付ける．例えばパス/usr/ast/mailboxは，ルートディレクトリにサブディレクトリusrが存在し，その中にさらにastというサブディレクトリが存在し，そこにファイルmailboxが含まれていることを示すものとなる．絶対パス名は，必ずルートディレクトリから始まり，一意な名前となる．

もう1つの方法は相対パス名(relative path name)である．これはワーキングディレクトリ(working directory)の概念と共に使用する．ワーキングディレクトリはカレントディレクトリ(current directory)とも呼ばれている．ユーザーは1つのディレクトリをカレント・ワーキングディレクトリとして定義する．ルートディレクトリから始まらないパス名は，ワーキングディレクトリから相対的に解釈される．UNIXではスラッシュで始まるパス名はすべて絶対パス名であり，その他はすべてユーザーのカレント・ワーキングディレクトリからの相対パス名となる．ワーキングディレクトリが/usr/astであれば，絶対パス名が/usr/ast/mailboxのファイルは，単にmailboxと指定するだけでよい．

5.2 ファイルシステムの設計

それではファイルシステムをユーザーではなく，システム設計者の側から考えてみることにする．ユーザーはファイルの命名法，認められている操作内容，ディレクトリ・ツリーの形態，その他インターフェイス関連の事項に関心があった．設計者はディスク空間の管理法，ファイルの保存法，効率よく，信頼性の高いシステムの構築法などが興味深い．以降の項では，問題点とそれを解決するために必要なトレードオフについて説明する．

5.2.1 ディスク空間の管理

ファイルは通常ディスクに保存されている．したがってディスクの管理はファイルシステムを設計する者にとって最大の関心事となる．nバイトのファイルを保存するためには，2つの方法がある．連続するnバイト上のディスク空間を割り当てる方法と，ファイルをいくつかの隣接(必ずしも隣接している必要はないが)ブロックに分割する方法である．同じような問題が純粋なセグメンテーションとページングの間のメモリ管理システムにも存在している．

ファイルを連続するバイトとして保存することは明らかに問題である．ファイルが成長した場合には，ディスク上の別な位置に移動しなければならない可能性があるからである．同じ問題はメモリ内のセグメントにも該当する．ただし，ファイルをディスク上で移動するのに比べ，メモリ内におけるセグメントの移動は高速である．

したがってほとんどのファイルシステムでは，ファイルを固定サイズのブロックに分割し，かつブロックが隣接している必要がない，という方法をとっている．ファイルを固定サイズのブロックに分割し保存する場合には，ブロックの大きさが問題となる．ディスクの構造から考えて，セクタ，トラック，シリンダなどが割当て単位として考えられる．ページング・システムではページサイズの利用も考えられる．

シリンダの様に大きな割当て単位を用いてしまうと，1バイトのファイルでもシリンダ全体を占有してしまうことになる．UNIXにおける平均ファイルサイズは約1Kであることが研究の結

果(Mullender Tanenbaum, 1984)判明している．したがって各ファイルに32Kのシリンダを割り当てることは，31/32，すなわち97%のディスク空間を無駄にすることとなる．逆に小さな割当て単位を用いると，各ファイルが多くのブロックから構成されることになる．そうすると，各ブロックの読出しは，シークと回転によるロスを生じるため，多くの小さなブロックから構成されるファイルを読み出す場合の速度が低下する．

一例として，トラックあたり32,768バイトのディスクを想定してみる．回転時間は16.67ミリ秒，平均シーク時間は30ミリ秒である．kバイトのランダムブロックを読み出すための時間をミリ秒で表すと，以下の様にシーク，回転による遅延，転送時間の合計となる．

$$30+8.3+(k/32768)\times 16.67 \quad (ms)$$

図5.4の実線は，この様なディスクのデータ転送効率をブロックサイズの関数として示したものである．すべてのファイルが1Kであると仮定した場合(測定平均サイズ)，図5.4の点線はディスク空間の効率を表す．空間を効率的に使用すると(ブロックサイズ<2K)，データ転送効率が落ちるという矛盾が生じることになる．時間的効率と，空間的効率は元来対立する関係にある．

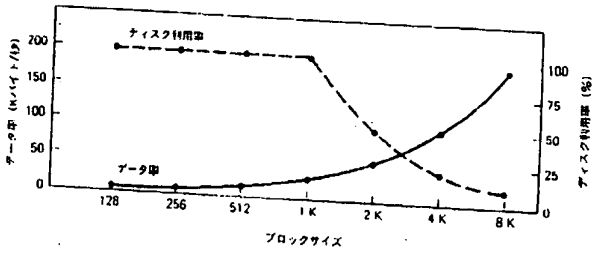

図5.4 ディスク空間利用率
実線(スケールは左)はディスクのデータ転送効率を示したものである．点線(スケールは右)はディスク空間の効率を示したものである．すべてのファイルは1Kとする．

通常の解決策は，512，1K，または2Kのブロックサイズを選択するという方法である．1Kのブロックサイズを512バイトのセクタサイズを持つディスクで選択した場合，ファイルシステムは必ず2つの連続するセクタを読み書きし，それを1つの分割不可能な単位として扱うことになる．

ブロックサイズを決定した後は，空きブロックの管理方法を決めなくてはならない．一般的には図5.5に示す2つの方法が用いられている．最初の方法はそれぞれが持ちうる限りの空きブロックの番号を格納しているディスクブロックの連結されたリストを用いるものである．ブロックが1Kであり，ディスクブロック番号に16ビット使用している場合，空きリスト上の各ブロ

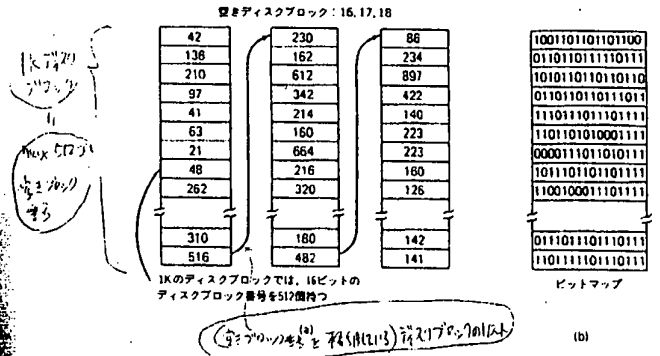

図5.5 空き領域の管理
(a)連結されたリスト上に空きリストを持つ (b)ビットマップ

クには，512個の空きブロック番号が格納されることになる．20Mのディスクでは20,000のディスクブロック番号をすべて保存するために最大40ブロックの空きリストが必要となる．

空き領域管理において，ビットマップを用いる方法もある．nブロックを持つディスク空間では，nビットのビットマップが必要となる．マップにおいて，空きブロックは1，また割り当てられたブロックは0で示される(またはその逆)．20Mのディスクでは，マップに対して20Kビット，すなわち3ブロックしか必要としない．ビットマップは1ブロックあたり1ビットを使用するだけでよい．連結されたリストの場合，1ブロックあたり16ビットが必要であったが，これと比較すると，かなり少ない空間で済むわけである．

ビットマップを保存できるだけの主記憶があれば，この方法が適している．ただし，メモリ内において，空きディスクブロック・リストを格納する領域が1ブロックしか確保できず，またディスクもほぼ満杯である場合，連結されたリストを使ったほうがよいだろう．メモリ内に1ブロック分のビットマップを置く場合，もし空きブロックがそのビットマップ内で検出できなければ，ディスクをアクセスし，残りのビットマップを読み出さなくてはならない．連結されたリストであれば，新しいブロックがメモリにロードされると，次のブロックを読み取るまでに，512のディスクブロックを割り当てることができる．

5.2.2 ファイルの保存

ファイルが連続するブロックから構成されている場合，ファイルシステムは各ファイルを構成するブロックを何らかの方法で管理しなくてはならない．ブロックを連続的に保存するという最も簡単な方法は，ファイルが成長していくため現実的ではない．事実，ファイルをブロックに分割せざるを得なかったのもこの問題に起因している．

ファイルのブロックを連結されたリストとして保存することは可能である．1,024バイトのディスクブロックに，1,022バイトのデータと，次のブロックを示すポインタを含める方法である．これには2つの問題がある．まずブロック内のデータバイト数が2の階乗ではなくなってしまうという問題が生じる．さらに深刻であるのは，ランダムアクセスの実行において多くの無駄が発生するということである．プログラムがファイル内の32,768バイト目にシークし，読み始めようとしている時，オペレーティング・システムは，32768/1022，すなわち33ブロックをシークし，必要なデータを検出しなければならない．シークを行うために33ブロックもの読取りを行うことは，効率的ではない．

しかしファイルを連結されたリストとして表現する際の問題は，ポインタをメモリ内に保存することによって解決できる．図5.6はMS-DOSに用いられている割当て手法である．この例では，ブロック6，8，4，2が割り当てられているファイルA，ブロック5，9，12が割り当てられているファイルB，そしてブロック10，3，13が割り当てられているファイルCを示している．

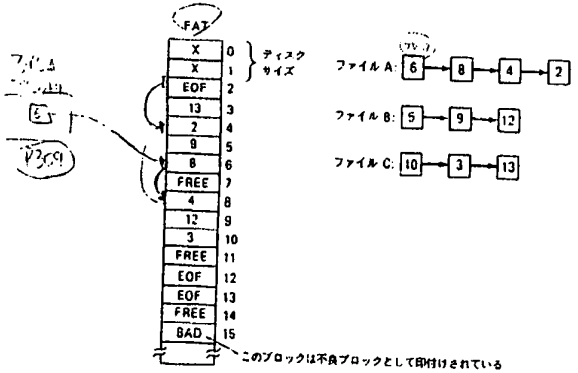

図5.6 MS-DOSで用いられている連結されたリストの割当て手法
エントリ0と1はディスクサイズの指定に用いられる．EOFはファイルの終端，FREEは空きブロックである．

各ディスクに用意されるテーブルを，ファイル割当てテーブル(FAT：File Allocation Table)と呼ぶ．1つのエントリが1ディスクブロックに対応している．各ファイルのディレクトリには，ファイルの先頭ブロックのブロック番号が格納されている．そのブロック番号に対応するFATのスロットには，次のブロックのブロック番号が格納されている．ファイルAはブロック6から始まり，FATのエントリ6はファイルAの次のブロック番号である8を格納している．FATエントリ8は次のブロック番号である4を格納している．エントリ4はエントリ2を，そしてエントリ2にはファイル終端(End of File)の印が付けられる．

この手法は，元来1Kのブロックサイズを用いた320Kのフロッピーディスク(MS-DOS標準)用に提供されたものであった．ブロック番号は12ビットであるため，320エントリのFATは480バイトを要することになり，512バイトのセクタにちょうど納まる．IBMがDOS 2.0から始めた，フロッピーディスクを360Kバイトとしてフォーマッティングするという方法では，FATが540バイトに増加した．この結果，1つのセクタには格納できなくなり，ディスクのレイアウトを変更してより大きな(2セクタ)FATを作成しなければならなかった．4,096よりブロック数の多いハードディスクが出現すると，12ビットのブロック番号は不適当となり，FATを再度変更しなければならなくなった．先見の明があれば，変更を繰り返す必要もなかっただろう．

また大容量のディスクにおいては，この手法を採用することが少なくなっている．1Kのディスクブロックを64,000個含んでいる64Mのディスクを仮定してみる(70Mのディスクであれば16ビット以上のブロックサイズが必要となり，さらに厄介なことになる)．FATは64K分のエントリ(各2バイト)を持ち，128Kを占有することになる．これをすべてメモリ内に常駐させると，かなりの量のメモリが消費されてしまう．しかしディスクに保存すると，ファイル内の32Kバイト目に対するシークを行う場合，ブロックの連鎖をたどるために1回以上33回以下のディスク読取りが必要となる．

FATの基本的な問題は，ディスク上のすべてのファイルに対するポインタが，1つのテーブル内にランダムに混在しているという点である．したがってたとえ1つのファイルをオープンする場合でも，FAT全体が必要になる．よりよい方法として，それぞれのファイルのためのブロックリストを別々の場所に保存しておく方法がある．UNIXはこの様な方法を採っている．

UNIXでは各ファイルに対応して，iノード(i-node)と呼ぶ小さなテーブルが用意されている．図5.7はこれを示したものである．このテーブルにはアカウント情報と，保護情報が含まれてい

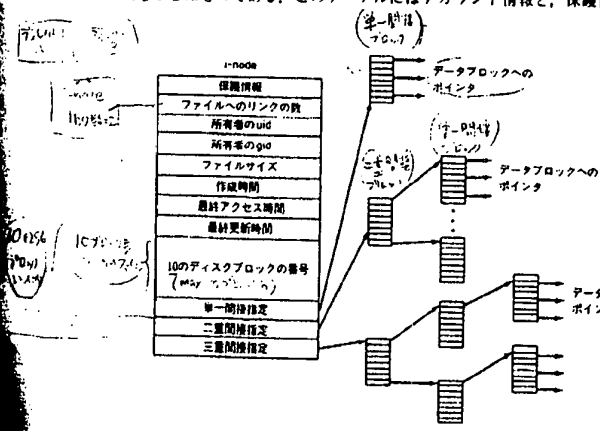

図5.7 iノード構造

る．これらに関しては後述する．ここでは10個のディスクブロック番号と，3つの間接ブロック番号に焦点をあてる．最長10ブロック分のファイルでは，すべてのディスクアドレスがiノードに保存されているため，ブロックの検出は簡単である．

ファイルが10ディスクブロック以上の大きさとなった場合，空きディスクブロックを獲得し，単一間接ブロック内のポインタがそれを指すよう設定される．単一間接ブロックは，ディスクブロックに対するポインタを保存するために用いられる．1Kのディスクブロックにおいて，32ビットのディスクアドレスを用いると，単一間接ブロックでは256個のディスクアドレスを保存することができる．この手法により，ファイルは最大266のブロックを持つことができる(iノード内に10個，単一間接ブロックに256個)．

ファイルの大きさが266ブロック以上になると，二重間接ブロック内のポインタが，最大256のポインタを持つディスクブロックを指すようになる．ただしこれらのポインタはデータブロックを指していない．256の単一間接ブロックを指すのである．二重間接ブロックは，ファイルの大きさが$266+(256^2)=65802$ブロックとなるまで使用できる．64M以上の大きさとなるファイルに関しては，三重間接ポインタを用いて，256の二重間接ブロックを指すポインタを含んだブロックを指す．

16Gバイトより長いファイルを扱うことはできない．もちろんディスクブロックの大きさを2Kにすれば，ポインタを持つ各ブロックは，256ではなく512個のポインタを持つことになり，最大ファイルサイズは128Gバイトとなる．128Gバイトディスク用のFATサイズは，予想するのもばかるほどだ．UNIXスキーマの利点は，間接ブロックが必要な場合にのみ使用されるというところである．10K以下のファイルの場合，間接ブロックは必要ない．最も長いファイルでさえも，最高3回のディスク参照によってファイル内の任意のバイトのディスクアドレスを得ることができる(iノードを得るためのディスク参照は除く．iノードはファイルがオープンされた時点でメモリ内に読み込まれ，ファイルがクローズされるまでメモリに保存される)．

MINIXの手法はUNIXのそれと同じであるが，iノードには7つのディスクブロック番号しか保存することはできず，三重間接ブロックも提供されていない点が異なっている．2バイトのディスクアドレスと，1Kブロックを使用した場合，最大262Mのファイルを扱うことができる．大半のパーソナルコンピュータに対しては，十分な大きさである．

5.2.3 ディレクトリ構造

ファイルの読出しを行うためには，それをオープンしなければならない．ファイルをオープンすると，オペレーティング・システムはユーザーが提供したパス名を用い，ディスクブロックの位置を検出，ファイルの読み書きに備える．パス名をiノードに対応付ける時(あるいはそれに相当するような操作)，ディレクトリ・システムがどの様な構造となっているか考えねばならない．ディレクトリ構造は簡単なものから，かなり高度なものまで様々である．

簡単なディレクトリ構造の一例として，CP/Mのディレクトリ・システム(Golden Pechura,

1986)を見てみることにしよう．図5.8はCP/Mのディレクトリを図示したものである．このシステムでは1つのディレクトリしか存在しないため，すべてのファイルはこのディレクトリを検索するだけで検出できる．エントリを発見すると，ディスクブロック番号も共に検出できることになる．ディスクブロック番号が，ディレクトリ・エントリ内に保存されているからである．ファイルが1つのエントリに格納しきれない数のブロックを割り当てられる時，そのファイルは追加ディレクトリ・エントリを割り当てられることになる．

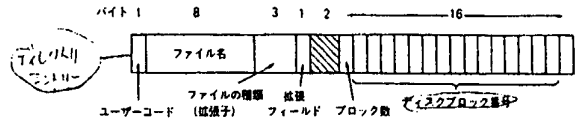

図5.8 ディスクブロック番号を持った各ファイルのディレクトリ

図5.8の各フィールドは以下の意味を持っている．ユーザーコードは，どのユーザーがファイルの所有者であるかを記録する部分である．検索時，現在ログインしているユーザーのエントリだけが対象となる．次の2フィールドには，ファイル名と拡張子が格納されている．拡張フィールドは，ファイルの大きさが16ブロック以上になった場合に複数のディレクトリ・エントリを割り当てなければならないために必要である．このフィールドには，ディレクトリ・エントリの順番に関する情報が格納される．ブロック数は，最大16のディスクブロック・エントリのうち，そのいくつが使用されているかを示す．残り16のフィールドにはブロック番号が保存されている．最後のブロックが満杯になっているとは限らないため，システムは最後のバイトまでの正確なサイズを決定することはできない(バイトではなく，ブロック単位でファイルサイズを記録しているため)．

次に階層化されたディレクトリ・ツリーを持つシステムの例を考えてみよう．図5.9はMS-DOSのディレクトリ・エントリを示したものである．長さは32バイトであり，ファイル名と先頭ブロック番号などが格納されている．先頭ブロック番号はFATに対するインデックスとして用いられ，それにより2番目のブロック番号を検出することになる．2番目以降のブロック検出も同

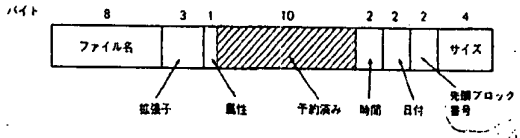

図5.9 MS-DOSディレクトリ・エントリ

様である．この様にして，特定のファイルのすべてのブロックを検出することができる．固定サイズを持つ(360K のフロッピーディスクでは 112 エントリ)ルートディレクトリを除き，MS-DOS のディレクトリはファイルとして扱われ，任意の数のエントリを持たせることができる．

UNIX や MINIX におけるディレクトリ構造は，図 5.10 の様に非常に簡単である．各エントリはファイル名とその i ノード番号だけを持っている．型，サイズ，時刻，所有者，およびディスクブロックに関する情報はすべて i ノード内に格納されている(図 5.7 参照)．UNIX 内のすべてのディレクトリはファイルであり，16 バイトのエントリを任意の数だけ持たせることができる．

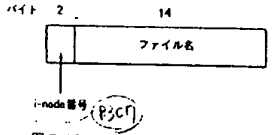

図 5.10 UNIX ディレクトリ・エントリ

ファイルをオープンすると，ファイルシステムは与えられたファイル名を用い，ディスクブロックを検出する．パス名/usr/ast/mbox の検出がどの様に行われるかを見てみることにしよう．これは UNIX における例であるが，階層化ディレクトリ・システムでは同様のアルゴリズムが用いられる．最初にファイルシステムがルートディレクトリを検索する．UNIX ではルートの i ノードはディスクの決まった場所に置かれている．

次にパスの最初の部分である usr を検出するため，ルートディレクトリを検索し，ファイル/usr に対する i ノードを得る．この i ノードから，システムは/usr のディレクトリを検出し，パス名の次の部分である ast を見つける．ast のエントリを検出すると，ディレクトリ/usr/ast の i ノードが得られたことになる．この i ノードからディレクトリ内容を取り出し，mbox を検索する．このファイルの i ノードはメモリに読み込まれ，ファイルがクローズされるまでメモリ内に置かれる．この検索の過程を図 5.11 に示している．

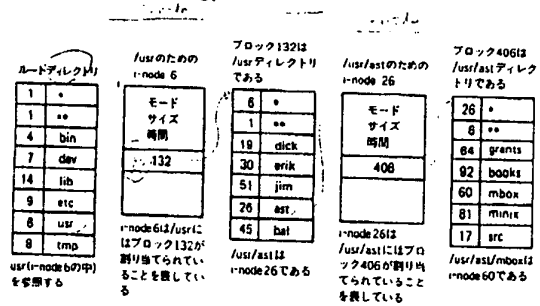

図 5.11 /usr/ast/mbox の検索過程

相対パス名の検索も，開始位置がルートではなく，ワーキングディレクトリであるという点を除けば，絶対パス名と同じである．各ディレクトリはディレクトリ作成時に割り当てられた "." と ".." に対するエントリを持っている．エントリ "." はカレントディレクトリに割り当てられている i ノード番号を保有しており，エントリ ".." は親ディレクトリに割り当てられている i ノード番号を保有しいる．したがって，../dick/prog.c の検索手続きにおいては，単にワーキングディレクトリの ".." を検索し，親ディレクトリの i ノード番号を検出し，そのディレクトリ内で dick を探すことになる．これらのパス名を扱うためにに，特殊な機構を用いる必要はない．ディレクトリ・システムにとって "." と ".." は，単なる通常の ASCII 文字列にすぎない．

5.2.4 共有ファイル

何人かのユーザーが同じプロジェクトに携わっている時，ファイルの共有が必要になる．例えばそれぞれのユーザーが所有しているそれぞれのディレクトリから，共有するファイルを同時に扱えると便利である．図 5.12 には，図 5.3(c)のファイルシステムを再度示している．ただし今回は，ユーザーC のファイルが，ユーザーB のディレクトリにも現れている．ユーザーB のディレクトリと共有ファイルの関係をリンク(link)と呼んでいる．ファイルシステム自身は，有向無周期グラフ(DAG：Directed Acyclic Graph)となり，もはやツリーではなくなる．

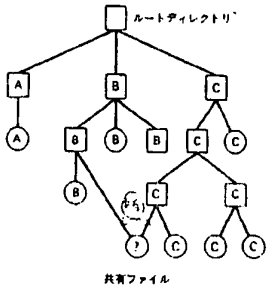

図 5.12 共有ファイルを持つファイルシステム

共有ファイルは便利ではあるものの，いくつかの問題がある．まず最初にディレクトリが CP/M の様に直接ディスクアドレスを持っている場合である．ファイルがリンクされた場合，ユーザーB のディレクトリにもディスクアドレスを持つことになる．もしユーザーB かユーザーC がこのファイルの後ろにブロックを追加したとすると，新しいブロックはどちらかのユーザーのディレクトリのみに保存されることになる．この変更は別のユーザーからは見ることができなくなり，つまりファイルの共有という目的は果たせないことになる．

この問題は，2つの方法によって解決できる．最初の方法は，ディスクブロックをディレクトリではなく，ファイル自身に対応した小型のデータ構造に持たせることである．これにより，ディレクトリはこのデータ構造だけを指すようになる．この方法はUNIXでも用いられている(その場合，小型のデータ構造に相当するものはiノードである)．

もう1つの方法では，システムに新しくLINK型のファイルを作成させることにより，ユーザーBがユーザーCのファイルの1つにリンクし，そのファイルをユーザーBのディレクトリに加える．新しいファイルにはリンクされたファイルのパス名だけが含まれている．Bがリンクされたファイルを読み取ると，オペレーティング・システムは読み取られているファイルがLINK型であると判断し，連結されたファイル名を検索し，そのファイルを読み取る．この様な方法はシンボリックリンク(symbolic linking)と呼ばれている．

いずれの方法にも欠点がある．最初の方法において，ユーザーBが共有ファイルにリンクする場合，iノードに記録されたファイルの所有者はCである．リンクによって所有者は変わらないが(図5.13参照)，iノードのリンク数は1増やされるため，システムはいくつのディレクトリ・エントリがそのファイルを指しているかが判断できる．

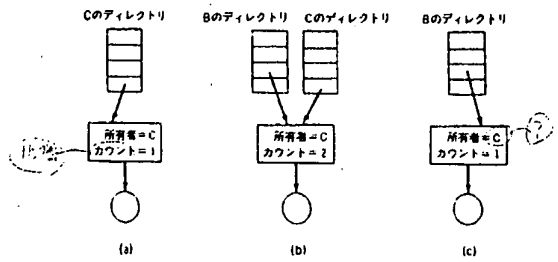

図5.13 DAGの欠点
(a)リンク前の状態 (b)リンク作成後 (c)最初の所有者がファイルを削除した後

その後ユーザーCがそのファイルを削除しようとした時点でシステムに問題が発生する．ファイルを削除し，iノードをクリアすると，Bは無効なiノードを指すディレクトリ・エントリを持つことになる．そのiノードが後に別のファイルに割り当てられた場合，Bのリンクは間違ったファイルを指すことになる．システムはiノード内のカウンタの値から，ファイルがまだ使用中であると判断することはできるが，削除のためにそのファイルに対するすべてのエントリを検出することは不可能である．ディレクトリを指すポインタをiノードに保存することもできない．なぜならディレクトリは無限に存在する可能性があるからである．

そこでiノードは保留し，カウントを1に設定し，Cのディレクトリ・エントリを除去してしまうという方法がある．図5.13(c)はこれを示したものである．これにより，ユーザーCが所有して

いるファイルのディレクトリ・エントリを持っているのはユーザーBのみとなる．システムがアカウント(CPU使用時間やディスク使用量などに対する課金)，またはクォータ(CPU使用時間やディスク使用量の割り当て，もしくは制限)を行っている場合，ユーザーCはユーザーBがそれを削除するまで，ファイルの料金を徴収されることになる．Bがこのファイルを削除した時点で，カウントは0になり，ファイルは本当に削除される．

シンボリックリンクでは，本当の所有者だけがiノードを指すポインタを持つことになるため，前述した問題は発生しない．このファイルにリンクしたユーザーは，iノードへのポインタではなく，パス名だけを与えられる．本当の所有者がファイルを削除すると，ファイルは破壊されてしまう．シンボリックリンクによるそのファイルへのアクセスは，システムがそのファイルを検出できなければ失敗に終わる．シンボリックリンクを削除してもファイルにはなんの影響もない．

シンボリックリンクの問題点は，余分なオーバーヘッドの発生である．そのパスを含んでいるファイルを読み出し，次にそのパスを解析し，iノードにたどり着くまで，1つ1つ追跡していかなくてはならない．これらの作業においては多くの余分なディスクアクセスが必要である．さらに各シンボリックリンクに対してiノードが1つ余分に必要になり，加えてパスを保存するための追加ディスクブロックも必要になる．ただしパス名が短い場合は，最適化のためにシステムはパス名をiノード内部に保存する．シンボリックリンクにはマシンのネットワーク・アドレスと，そのマシン上のパス名を提供するだけで，世界中のどのマシンのファイルにもリンクできるという利点がある．

またシンボリックリンクやその他のリンクにより，別な問題が発生する．リンクが認められると，ファイルは複数のパスを持つ可能性がある．特定のディレクトリから始まり，そのディレクトリ内のすべてのファイルとサブディレクトリを検出するというプログラムは，リンクされたファイルを複数回検索することになる．例えばディレクトリとそのサブディレクトリのすべてのファイルをテープにダンプするプログラムは，リンクされたファイルのコピーを複数回行うことになる．さらにテープを別のファイルに読み出すと，よほど利口なダンププログラムでない限り，リンクされたファイルがもとのとおりリンクされるのではなく，ディスクに2回コピーされる．

5.2.5 ファイルシステムの信頼性

ファイルシステムの破壊は，コンピュータの破壊よりはるかに悲惨な結果を招くことが多い．例えばコンピュータが火災，落雷，あるいはキーボードの上にこぼしたコーヒーによって破壊されると，費用はかかるが交換品を購入すればことは済む．安価なパーソナルコンピュータの場合，販売店に持ち込めば，数時間で交換してくれるだろう(ただし大学の様に注文書の発行までに，いくつもの委員会の承認を得なければならない場合は別である)．

コンピュータのファイルシステムが，ハードウェア，ソフトウェア，フロッピーディスクをかじったネズミの仕業などで，取り返しのつかない状態に陥った場合，すべての情報を復元することは困難であり，時間もかかり，しかもたいていの場合は不可能である．プログラム，書類，顧

客ファイル，租税記録，データベース，営業計画，その他のデータを永久に失ってしまった者にとって，その痛手は図り知れないほど大きいものである．ファイルシステムにおいて，装置やメディアに対する物理的な保護を行うことはできないが，情報を保護することは可能である．この項では，ファイルシステムの安全保護に関していくつかの説明を行う．

3章でも説明したように，ディスク内に不良ブロックが存在することも珍しくはない．フロッピーディスクは出荷時において完璧な状態であっても，使用中に不良ブロックが発生するかもしれない．ハードディスクでは，初めから不良ブロックが存在していることが多い．不良ブロックのないディスクを製造するのは困難である．事実，大半のハードディスク製造元は各ドライブに，検査中に発見した不良ブロックリストを付けている．

不良ブロックに対しては2つの解決方法が取られている．1つはハードウェアの面から，そしてもう1つはソフトウェアの面からの解決法である．ハードウェアによる解決策は，ディスク上の1つのセクタを不良ブロックリスト専用に使用する方法である．コントローラが最初に初期化された時，不良ブロックリストを読み出し，交代ブロック(またはトラック)を不良ブロックと置き換え，その内容を不良ブロックリストに記録する．これによって不良ブロックに対するすべての要求では，交代ブロックが用いられることになる．

ソフトウェアによる解決策では，ユーザーまたはファイルシステムがすべての不良ブロックを含んだファイルを入念に作成しなくてはならない．これによって不良ブロックは空きブロックから除外され，データファイルに現れることはない．不良ブロックからファイルが読み書きされない限り問題は発生しない．ディスクのバックアップを行う時，誤ってこのファイルを読み出さないよう注意しなければならない．

■ バックアップ

不良ブロックを上記の方法で入念に処理したとしても，ファイルを頻繁にバックアップしておくことは大切である．重要なデータブロックが破壊されてから自動的に交代トラックに切り換えるという方法は，競争馬が逃げてから，馬小屋の鍵をかけるようなものである．

フロッピーディスク上のファイルシステムは，そのフロッピーディスク全体をブランクのディスクにコピーするだけでバックアップできる．小型のハードディスクのファイルシステムはディスク全体を，標準の9トラック磁気テープ(1巻あたり50M保存できる)か，ストリーマ・テープ(各種のサイズが提供されている)のいずれかにダンプすることにより，バックアップできる．

大型のハードディスク(例えば500M程度のもの)になると，ドライブ全体をテープにバックアップするのは時間もかかり，効率が悪い．実行は高速でも容量の半分を無駄にしてしまうことになるが，各コンピュータに2つのドライブを接続するという方法がある．ドライブそれぞれを，データ保存用とバックアップ用に2分して使用するのである．毎晩ドライブ0のデータ部分がドライブ1のバックアップ部分にコピーされ，同時にその逆も行われるといった具合である．図5.14はこれを示したものである．この方法により，たとえ1つのドライブが完全に破壊されても，

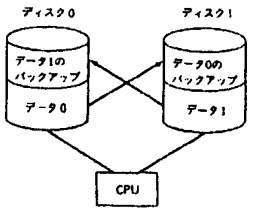

図5.14 各ドライブを別のドライブにバックアップする容量の半分が浪費されることになる．

情報は保存されることになる．

毎日ファイルシステム全体をダンプする代わりに，増分ダンプ(incremental dumps)という技法が用いられている．増分ダンプの最も簡単な形態は，完全なダンプを週単位や月単位で周期的に行い，この周期的なダンプ以降修正されたファイルのみ日々のダンプを行うというものである．またこれより優れている手法としては，前回ダンプした以降に変更されたファイルのみをダンプするというものがある．

この技法を実現するためには，各ファイルのダンプ時刻のリストをディスク上に保存していなければならない．前回のダンプ以降に変更されている場合は，再度ダンプされ，最終ダンプ時刻が現時刻に変更される．月単位でこれを実行すると，この方法は日々のダンプ用に31本のテープと，月1回行われる完全なダンプを保存できるだけのテープ数が必要となる．この他，これより少ない数のテープを使うという，もっと複雑な手法も用いられている．

■ ファイルシステムの一貫性

もう1つ，信頼性が問題になるのは，ファイルシステムの一貫性である．大半のファイルシステムでは，ブロックを読取り，それを修正し，後でそれを書き込む．修正されたブロックが書き込まれる前にシステムがクラッシュしてしまうと，ファイルシステムは一貫性を失ってしまう．これは書き込まれていないブロックのいくつかが，iノードブロック，ディレクトリ・ブロック，あるいは空きリストを含んでいるブロックである場合は特に深刻な問題となる．

一貫性のないファイルシステムを扱うために，大半のコンピュータではファイルシステムの一貫性を検査するユーティリティ・プログラムを持っている．ファイルシステムがブートされた時や，特にクラッシュの後ではこのプログラムを必ず実行する．以下はUNIXやMINIXにおいてこの様なユーティリティがどの様な働きをするかを説明したものであるが，他の多くのシステムでも同じような動きが見られる．ファイルシステム検査プログラムは，各ファイルシステム(ディスク)を個別に検査する．

ブロックとファイルに対し，2種類の一貫性検査が行われる．ブロックの一貫性を検査するため

に，プログラムはブロックあたり2つのカウンタを持つテーブルを作成する．カウンタは当初どちらも0に設定されている．最初のカウンタはファイル内に何度ブロックが現れたかを記録するものである．2番目のカウンタは空きリストに何回現れたかを記録するものである(または空きブロックのビットマップ)．

次にプログラムはすべてのiノードを読み取る．iノードを検査すれば，該当するファイルで使用されているブロック番号のリストを作成できる．各ブロックが読み取られた後，最初のテーブルの各カウンタが増加される．次にプログラムは，空きリスト，またはビットマップを検査し，使用されていないブロックをすべて検出する．空きリストにブロックが発見されるたびに，2番目のテーブルのカウンタが増加される．

ファイルシステムに一貫性のある場合は，図5.15(a)に示すように，各ブロックがどちらかのテーブルに1のカウントを持っている．しかしクラッシュが発生すると，テーブルは図5.15(b)の様になってしまう．ここではブロック2はいずれのテーブルにも見られない．これは紛失ブロック(missing block)として報告される．紛失ブロックそのものは特に害はないが，ディスク空間を浪費するため，ディスク容量が減少してしまう．紛失ブロックの対処法は簡単である．ファイルシステム検査プログラムがそれを空きリストに入れるだけである．

図5.15(c)の様な状況も考えられる．ここではブロック番号4が，空きリストで2回発生しているのがわかる(空きリストが実際にリストである場合に限り重複が起こる．ビットマップの場合はありえない)．解決策は簡単である．空きリストを再構築すればよい．

最悪な事態として予想されるのは，同じデータブロックが図5.15(d)のブロック5の様に，複数のファイルに割り当てられてしまうことである．ファイルのうち，いずれかが削除されると，ブロック5は空きリスト上に配置され，同一ブロックが同時に使用中と，空き状態の両方の状態になってしまう．両方のファイルが削除された場合，そのブロックは空きリストに2回入れられることになる．

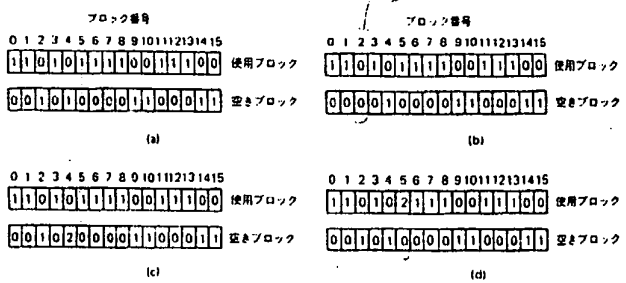

図5.15 ファイルシステムの状態
(a)一貫性が保たれた状態 (b)紛失ブロック
(c)空きリストにおける重複データブロック (d)重複データブロック

ファイルシステム検査プログラムは，ここで空きブロックを割り当て，ブロック5の内容をそこにコピーし，そしてコピーしたものをファイルの1つに挿入する．この様にしてファイルの持つ情報はそのままにし(とはいってもその内容は変わってしまっているだろうが)，ファイルシステム構造の一貫性だけが保たれる．ユーザーが損傷内容を検査する際の参考とするために，エラーが報告される．

各ブロックが正しく利用されているかという検査のほかに，ファイルシステム検査プログラムはディレクトリの検査も行う．ここでも2つのカウンタのテーブルを用いるが，ブロックごとではなく，ファイルごとに使用する点が異なっている．ルートディレクトリから始まり，再帰的にツリーを下り，ファイルシステム内の各ディレクトリを検査していく．各ディレクトリ内のファイルごとに，そのファイルのiノードカウンタを増加していく(ディレクトリ・エントリの構造に関しては，図5.10を参照)．

これらの作業が完了すると，iノードを指しているディレクトリ・エントリの数を示した，iノード番号をインデックスとするリストができ上がる．そしてこれらの番号とiノード自身に保存されたリンク数とを比較する．一貫性のあるファイルシステムでは，この2つが一致する．しかしiノード内のリンク数の方が大きい，または小さい場合には，何らかの誤りがある．リンク数がディレクトリ・エントリの数以上の場合，ディレクトリ内のすべてのファイルが削除されても，カウントが0以外となり，iノードが削除されないことになる．この誤りはそれほど深刻ではないが，どのディレクトリにも存在しないファイルを持つことになり，ディスクの空間を浪費することになる．iノード内のリンク数を正しい値に設定することにより，これが解決できる．

リンク数がディレクトリ・エントリの数以下である場合は，非常に危険である．2つのディレクトリ・エントリがファイルに連結されたが，iノードでは1つしか存在しないことになっていれば，どちらかのディレクトリ・エントリを削除することによって，iノード内のリンク数は0になってしまう．i-node内のリンク数が0になった時点で，ファイルシステムはそれを未使用として記し，そのブロックをすべて解放する．この結果，あるディレクトリは未使用のiノードを指してしまうことになり，解放されたブロックは他のファイルに割り当てられてしまう．これも，iノード内のリンク数をディレクトリ・エントリの実際の値に修正すれば，解決できる．

ブロックとディレクトリの検査という2つの操作は，システムの効率を上げるためしばしば統合されて行われる(すなわちiノード上では1回のパスだけでよい)．これ以外にも検査を行うことができる．例えばディレクトリは，iノード番号とASCII名を含む特定の形式を持っている．iノード番号がディスク上のiノードの数よりも大きければ，ディレクトリが破壊されていることがわかる．

さらに各iノード内の保護情報型が，0007の様に所有者にもユーザーグループにも全くアクセスを認めず，第3者にはファイルの読み書きや実行を認めるといった，奇妙なものもある．少なくとも，第3者に対して所有者よりも多くの権利を与えているようなファイルを検出させることには意義があるだろう．例えば1,000以上のエントリを持つディレクトリも疑ったほうがよいだ

ろう．ユーザーディレクトリに存在するが，所有者がスーパーユーザーで，SETUIDビットがオンとなっているファイルは，その機密保持状態に問題のある場合が多い．少し工夫すれば，奇妙な属性を持つファイルを長いリストにして報告することができる．

以上，ユーザーをクラッシュから保護する際の問題点について説明してきた．ファイルシステムによっては，ユーザーの誤操作などを考慮したユーザーに対する保護が必要な場合もある．ユーザーが以下の入力を行い，.o で終了するすべてのファイル(コンパイラの作成したオブジェクトファイル)を削除しようとした．

```
rm *.o
```

ところが誤って以下の様に入力してしまったとしよう(アスタリスクの後ろにスペースがある点に注意せよ)．

```
rm * .o
```

これにより，rm はカレントディレクトリ中のすべてのファイルを削除し，.o がないというエラーメッセージを表示することになる．MS-DOS，およびあるシステムでは，ファイルを削除すると，単にビットがディレクトリに設定されるか，ファイルの削除を示す印がiノードに付けられるかの，いずれかである．実際にそれが必要になるまで，ディスクブロックは空きリストには戻されない．したがってユーザーが即座にエラーを発見すると，特殊なユーティリティ・プログラムを実行して削除したファイルを取り戻す(復元する)ことができる．

5.2.6 ファイルシステムの性能

ディスクへのアクセスは，メモリへのアクセスよりもかなり低速である．メモリワードの読み出しは通常，多くても数百ナノ秒も要さない．ところがディスクブロックの読み出しには，10ミリ秒も必要である．メモリの1万倍も，速度が低下する．この様なアクセス時間の違いにより，多くのファイルシステムはディスクアクセス数をできるだけ抑えるように設計されている．

ディスクアクセスを削減するために多用されている技法には，ブロックキャッシュ(block cache)またはバッファ・キャッシュ(buffer cache)と呼ばれる技法がある(キャッシュはフランス語のcacherに由来する言葉で，隠蔽の意を持つ)．ここでのキャッシュとは，論理的にはディスクに属しているが，性能上メモリに保存されているブロックの集合を意味する．

各種のアルゴリズムを用いてキャッシュを管理することができるが，最も一般的なものは，すべての読出し要求を調べ，必要なブロックがキャッシュに含まれているかどうかを確認する方法である．含まれている場合には，ディスクにアクセスせずに，読取り要求を満たすことができる．そのブロックがキャッシュ内に存在しなければ，最初にキャッシュ内にそれを読み取り，そのブロックを必要とする部分にコピーする．以後，同一ブロックに対する要求があった場合には，キャッシュを用いればよい．

空きがなくなっているキャッシュにブロックを読み取る場合には，いずれかのブロックをキャッシュから削除しなければならない．もしキャッシュに置かれてから何らかの変更が加えられている時は，ディスクに書き込まなければならない．これはページングと非常に似ており，4章で説明した通常のページング・アルゴリズム，すなわちFIFO，セカンドチャンス，LRUなどがすべて適用される．ページングとキャッシングの主な違いは，キャッシュの参照回数が比較的少なく，すべてのブロックを正確なLRU順に，連結されたリストとともに保存していくことがそれほど困難ではないという点にある．

残念ながら，ここに落とし穴がある．正確なLRUが可能な状況にあって，LRUを使用することが望ましくないことがわかった．問題は先の項で触れたクラッシュとファイルシステムの一貫性に関係がある．iノードブロックの様に重要なブロックがキャッシュに読み込まれ，修正されたにも関わらず，ディスクに再度書き込まれなかったとすれば，クラッシュによってファイルシステムの一貫性は失われてしまう．iノードブロックをLRUチェーンの最後に配置したら，先頭にたどり着いてディスクに再度書き込まれるまで，かなりの時間を要することになってしまう．

さらに二重間接ブロックに該当するブロックでは，短い間隔で2回参照されることは滅多にない．これらを考慮してLRUスキーマを修正してみる．特に，以下の点を配慮する．

- そのブロックはすぐに再度必要とされるか
- ファイルシステムの一貫性を保つために，そのブロックは必要か

これらの質問に対し，ブロックはiノードブロック，間接ブロック，直接ブロック，全体が使用されているデータブロック，部分的に使用されているデータブロックに分類される．すぐに再度必要とはされないブロックはLRUの終わりではなく，先頭に配置され，そのバッファが即座に再利用されるようにする．書き込まれようとしている部分使用のブロックなど，すぐに必要となるブロックは，リストの最後に置かれ，長時間保存されるようにする．

2番目の質問は最初の質問とは関連していない．ブロックがファイルシステムの一貫性を保つために必要なもので(基本的にはデータブロック以外ならすべてこれに当たる)，更新されたものであれば，LRUリストのどこに置かれていても，即座にディスクへ書き込まなければならない．重要なブロックを即座に書込むことにより，クラッシュによる被害を最小限にとどめることができる．

この様に，ファイルシステムの一貫性を保つための方法を用いても，実際に書き込むまでデータブロックを，キャッシュに長期間とどめておくことは望ましくない．パーソナルコンピュータを使って本を書いている人を例にとってみよう．彼が何度もエディタに編集用のファイルをディスクに書き込むよう指示したとしても，まだディスクには何も保存されておらず，キャッシュにすべて残っている可能性がある．システムがここでクラッシュしてしまうと，ファイルシステムの構造は破壊されないが，その日1日の作業内容が失われてしまう．

この様なことが頻繁に発生すると，ユーザーは作業意欲を失ってしまうだろう．この様な問題

を解決するため，システムは2つの技法を提供している．UNIXではシステムコールSYNCが用意されており，SYNCでは更新されたすべてのブロックを即座に強制的にディスクに書き込む．システムを起動すると，通常updateと呼ばれるプログラムがバックグラウンドで立ち上がり，システムコールSYNCを実行し，次の呼出しまで30秒間休止するという無限ループを実行する．結果として30秒以上前に行われた作業が，クラッシュによって失われることがなくなる．

MS-DOSでは，ブロックが更新されると，それをすぐにディスクに書き込んでいる．修正されたすべてのブロックが即座に書き込まれる方式のキャッシュは，ライトスルーキャッシュ(write through cache)と呼ばれている．これは一般のキャッシュよりもかなり多くのディスク入出力操作を必要とする．2つの方法の相違は，プログラムが1Kブロックが満杯となるまで，1文字ずつ書き込んだ時に明らかになる．UNIXはキャッシュ内にすべての文字を貯めておき，そのブロックを30秒に1回ずつ，またはブロックがキャッシュから削除されるたびに書き込む．MS-DOSは書き込まれた文字すべてに対してディスクアクセスを行う．もちろん大半のプログラムは内部バッファリングを行い，1文字ずつではなく，行やそれより大きな単位でWRITEシステムコールにより書込みを行う．

この様なキャッシング技法の違いにより，UNIXシステムからSYNCを行わずに(フロッピー)ディスクを取り出すと，たいていの場合データの紛失を招くだけでなく，しばしばファイルシステムの破壊にまで至ってしまう．MS-DOSではこの様な問題は起こらない．この様な異なった方法が取られている背景には，UNIXは，すべてのディスクが取りはずしできないハードディスクであるという環境で開発され，MS-DOSは，フロッピーディスク環境から始まったという背景がある．ハードディスクがマイクロコンピュータでさえも一般的になるにつれ，より効率の高いUNIXの技法が多用されるようになるだろう．

キャッシングだけがファイルシステムの性能を向上させる方法ではない．もう1つ重要な技法として，連続アクセスを行うブロックをできれば同じシリンダ内の近接する場所に置くことにより，ディスクアームの移動量を削減する方法がある．出力ファイルが書き込まれると，ファイルシステムはブロックを1つずつ必要に応じて割り当てる．空きブロックがビットマップで管理されていれば，直前のブロックに最も近い空きブロックを検出することは非常に簡単である．空きリストを用いており，その一部がディスクに存在する場合は，近接したブロックの割当てはかなり困難になる．

しかし空きリストを使っても，ある程度のブロッククラスタリングを行うことは可能である．その際はディスクの保存内容をブロックではなく，連続するブロックグループで記録していくのがこつである．1トラックに512バイトのセクタが64個含まれているとすれば，システムは1Kブロック(2セクタ)を使用するが，ディスクの保存容量の割当ては2ブロック単位(4セクタ)で行う．これは2Kのディスクブロックを使用するのとは異なる．キャッシュも1Kのブロックを用いて，ディスク転送も引き続き1K単位で行われるが，ファイルをアイドル状態のシステムで連続的に読み出す場合は，シーク回数は2分の1に減少し，性能をかなり向上させることができる．

同じ技法に若干変化を加え，回転配置を考慮したものがある．ブロックを割り当てる時，システムはファイル内の連続ブロックを同じシリンダに配置しようとするが，最大スループットを得るためにインターリーブされる．したがってディスクの回転時間が最大16.67ミリ秒であり，ユーザープロセスがディスクブロックを要求してから，それを入手するまで4ミリ秒かかる場合は，各ブロックは少なくとも前のブロックから1/4の距離をおいて配置されなくてはならない．

さらに，iノードもしくは，それに相当するものを用いているシステムにおいて，性能向上を妨げるのは，短いファイルを読み出す場合でもiノードとブロックに対して合計2回のディスクアクセスが必要になるという点である．通常のiノード配置を示したのが図5.16(a)である．ここではすべてのiノードがディスクの先頭近くに配置されており，iノードとブロックの平均距離がシリンダの約半分であるため，シーク距離も長くなる．

ここでiノードをディスクの先頭ではなく，中央に置き，iノードと最初のブロックとの距離を2分の1に削減することによって，容易に性能を向上させることができる．図5.16(b)に示されたもう1つの方法は，ディスクをシリンダグループに分割し，それぞれのグループにiノード，ブロック，空きリストを持たせるという技法である(McKusick 他，1984)．新しいファイルを作成する時，任意のiノードを選択することはできるが，その際，iノードと同じシリンダグループ内にブロックを検索する．使用可能なブロックがなければ，近接のシリンダグループ・ブロックが使用される．

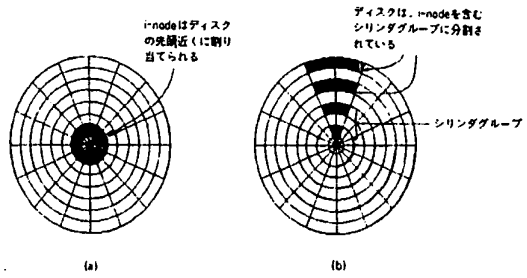

図5.16 iノードの位置
(a)ディスクの先頭に置かれたiノード
(b)シリンダグループに分割されたディスク．それぞれが独自のブロックとiノードを持っている．

5.3 ファイルサーバー

分散処理システムには，他のマシンにファイル機能を提供するマシンが存在していることが多い．これらのマシンはファイルサーバー(file servers)と呼ばれている(Birrell Needham, 1980；Fridrich Older, 1981；Svobodova, 1984；Swinehart 他, 1979)．分散処理システムのコストを抑えるための一般的な方法として，ユーザーにディスクレス・ワークステーションを与え，共通のファイルサーバーにREADおよびWRITE要求を送ることにより，ネットワーク経由でそのファイルにアクセスさせるという方法がある．

原則的に，この様なファイルサーバーは，UNIX，MS-DOSあるいは他の一般的なファイルシステムと全く同じインターフェイスを持つことができる．実際には研究者がいろいろな新しいアイデアを試験し構築した．結果として多くのファイルサーバーがこれまでのファイルシステムには見られなかった機能を持つようになった．以下の項ではそのうちいくつかを見ていくことにする．

5.3.1 インターフェイス・レベル

ファイルサーバーは以下の3つのレベルのいずれにおいてもユーザー(クライアント)にインターフェイスを提供することができる．最も簡単なものは遠隔ディスク(remote disk)である．このモデルでは，各ユーザーに対してファイルサーバーのディスク上に専有部分を仮想ディスク(virtual disk)として割り当てる．ユーザーはローカルディスクと同様に仮想ディスクを扱うことができる．ファイルサーバーはコマンドREAD BLOCKとWRITE BLOCKを，ローカルディスクと同じように提供する．

次は，ファイルサーバーはファイルだけを持ち，ディレクトリは持たないという方法である．ファイルの作成，削除，読出し，書込み，およびシークを行うコマンドが提供されている．ユーザーがファイルを作成すると，ファイルサーバーは以降の操作に用いる識別子を返してくる．識別子には例えばランダムな長い番号など，不正ユーザーが識別しにくいものを用いる．ASCII名をファイルサーバーの識別子に関連付けるためのディレクトリを設けるか否かは，ユーザー自身が決定する．この様なディレクトリを設ける場合は，識別子にはUNIXのディレクトリに保存されているiノード番号に類似したものを用いることもできる．

このスキーマにおける1つの問題は，ユーザーがファイルサーバー上にファイルを作成し，ディレクトリに識別子を書き出す前にシステムがクラッシュした場合，ファイルが「紛失」してしまう点にある．永久に存在はするが，識別子がわからないためアクセスができなくなる．この様な状態から脱出する唯一の方法は，ユーザーが自身のファイルの全リストを要求できるようなコマンドをファイルサーバーに設けることである．

3番目のインターフェイス・レベルは，ファイルサーバーにUNIXの様な完全なファイルシステムを持たせつつ，さらにより高度な機能を実現するという方法である．この方法を用いると，コマンドはファイルの操作だけでなく，ディレクトリの作成，削除，ワーキングディレクトリの変更，既存ファイルへのリンク作成と解除なども行えるようになる．リモート・ファイルサーバーが完全なディレクトリ機能を提供する時，ユーザーのワークステーションがリモート・ファイルシステムをワークステーションのファイルツリーにマウントできることもある．その様な場合，マウント後はリモートファイルへのアクセスが，ローカルマシン上のルートからの絶対パス名を指定するか，またはカレント・ワーキングディレクトリ(遠隔ディスク上にあると考えられる)からの相対パス名を指定するだけで行えるようになる．したがってそれがリモート・ファイルシステムであるということを意識する必要もなくなる．

5.3.2 アトミック更新

高度な技術が産み出したファイルシステムは，結果としてある意味ではかつてのものほど信頼性がないことが判明した．その昔，まだディスクが発明されていない時代に，企業がとっていた在庫管理方法を考えてみよう．通常マスターテープに全製品のリストと，その在庫数を記載していた．たいていはオリジナルのテープがほこりや，湿気，テープドライブによる破損によって万が一解読不能になった場合を考えて，バックアップ用のコピーがとられ，同じように保管されていた．

毎日一度マスターテープがドライブ1に着装され，その日の売上を記録したテープがドライブ2に，そしてブランク・テープがドライブ3に着装された．そして更新用プログラムを実行し，マスターと売上テープを読み出し，新しいマスターテープを作成した(そしてすぐにバックアップ用のコピーが取られた)．このプログラムが途中でクラッシュしてしまっても，3つのテープを巻き戻して，最初からやり直せばよかった．

このシステムには更新が問題なく完了しても，失敗に終わっても，オリジナルのテープには何の影響も及ぼさないという利点がある．さらにマスターテープが破損しても，いつでもバックアップ・コピーを用いることができる．

ディスクが発明されると，当然のことながらマスターテープは，ディスク上のファイルとなった．マスターテープの更新は，このファイルを読み出して更新する作業に変わった．唯一の問題は，更新中にクラッシュしてしまった場合，システムを復旧し，もとの状態に戻し，更新プログラムを再実行することが不可能な点であった．すでにどれだけのレコードが更新されているか判断できないからである．

3つのテープを用いたシステムには見られたが，ディスクを用いた更新においては見られなかった性質は，アトミック更新(atomic update)または故障復元性(failure atomicity)と呼ばれている．すなわちレコードに対する更新は完全に成功するか，しない場合にはシステムはもとのままのいずれかである．アトミック更新が失敗に終われば，更新用プログラムが再度実行されるだけである．回避しなくてはならないのは，データの全部ではなく，一部だけを変更するような

更新を行うことにより，ファイルを部分的に更新しただけの，不明瞭な状態にしてしまうことである．

テープシステムの性質でもう1つ注目すべき点は耐故障性(fault tolerance)と呼ばれるものである．マスターテープに解読できない場所があっても，バックアップをとってあるため心配する必要はなかった．理論的にはファイルシステムは，すべてのファイルのコピーを2つ保存することができるが，実際に2つコピーをとるシステムはほとんどない．

耐故障性を提供しているファイルサーバーでは，通常図5.17の様にアトミック更新によって，2つの物理ドライブから論理ディスクドライブを実現することになる．論理ブロックnに情報が書き込まれた後，まずサーバーはそれをドライブ1上の物理ブロック上に書き込む．そしてそれを読み返し，正しく書き込まれたかを確認する．何も問題がなければ，サーバーは同じ情報をドライブ2上の物理ブロックnに書き込み，それを検証する．

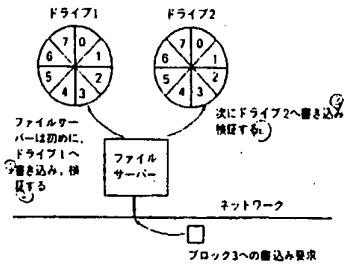

図5.17 ファイルサーバーの耐故障性の実現
同じ情報を持つ2つのディスクドライブを使用することによって，安定した保存を行うことができる．

この技法は安定記憶機構(stable storage)と呼ばれており(Lampson Sturgis, 1979)，興味深い性質を持っている．まず不良ブロックが偶然どちらかのブロックにできたとしよう．周期的に，例えば毎晩ファイルサーバーは不良ブロック，すなわちチェックサム・エラーを持つブロックを検出するために，両方のディスクを読み取る．不良ブロックを検出した場合，他のドライブのコピーを，不良ブロックに上書きする．それによってエラーを回避することができる(ブロックが物理的に破損している時，サーバーはそのためにあらかじめとっておいた交代トラックを用いることができる)．2つのブロックが同時に不良になってしまう可能性が非常に低いと仮定すると，頻繁にディスクエラーが発生したとしても，安定記憶機構によるデータ紛失は考えられない．

それではサーバークラッシュの影響について考えてみよう．どちらかのドライブに書込みを行っている間にクラッシュが発生すると，書き込まれているブロックはチェックサム・エラーを引き起こす．このエラーは必ず検出されるため，正常なブロックを使って不良ブロックを置き換えるとよい．サーバーがドライブ1に書き込んでいる間にクラッシュした場合，システムはもとの状態に復元される．ドライブ2に書き込んでいる最中にクラッシュすると，システムは更新された状態に復元される．安全記憶機構に対する書込みが成功しようと，失敗しようと，あいまいな状態になってしまうことはない．

アトミック更新にやや関連しているのがマルチバージョン・ファイル(multiversion files)である．ファイルサーバーがマルチバージョン・ファイルをサポートしている場合，いったん作成された後には，二度と再びファイルに対する更新は行われない．その代わりファイルの一時的なコピーを作成し，それをアトミック更新によって変更し，そのファイルを最新のバージョンとする．したがってファイルは時系列の不変のバージョンから構成されている．ファイルの読取り要求では，特に明示されていない限り，必ず最新のバージョンを読み出す．

5.3.3 同時実行制御

典型的なファイルシステムでは(例えばUNIX)，2人のユーザーが同時に1つのファイルを更新すると，コマンドREADとWRITEは到着順に実行される．しかし例えば銀行で，2人の顧客がそれぞれ同じ口座に同時に振込みを行おうとしたらどうなるだろう．その口座には当初500ドルとなっており，顧客はそれぞれ200ドルと300ドルを振り込もうとしている．以下の様な取引内容となるだろう．

①顧客1のプログラムは残高を読み出し，500ドルであることを知る．
②顧客2のプログラムは残高を読み出し，500ドルであることを知る．
③顧客1のプログラムは残高を500＋200＝700に更新する．
④顧客2のプログラムは残高を500＋300＝800に更新する．

最終的な残高は800ドルになる．顧客1が顧客2より遅く処理されれば，最終的な残高は700ドルになっていただろう．どちらにせよ2つの更新が同時に行われているため，最終的な結果は誤ったものとなる．ここで必要なのは，最初の顧客がプログラムを実行し，次にもう1人の顧客が実行するといった方法で，上記の様な状態が発生しないようにすることである．

同時に更新が行われても，任意の順序で更新を行った場合と同じ結果となるようにすることを直列化(serializability)と呼んでいる．また直列化のための技法を同時実行制御アルゴリズム(concurrency control algorithms)と呼んでいる．アトミック更新と同じように，データベースシステムではすでに普及しており，今後はファイルサーバーにも採用されることになるだろう．

多くのファイルサーバーが同時実行制御技法として，なんらかの形態でロック(locking)機能をユーザーに提供している．あるクライアントがファイルをロックすると，他のクライアントがそのファイルを使用，またはロックしようとしてもサーバーが拒否する．ロックを使い，次の様な方法で同時実行制御を行う．更新を実行する前に必要なファイルをすべてロックする．もしいずれかのファイルがすでにロックされていれば(他のクライアントによって)，ロックしたファイル

をすべてロック解除し，何も変更されない状態で実行が失敗に終わる．すべてのファイルのロックが成功した場合には，ファイルの読み書きが行われ，ロックは解除される．

ロッキングにおける重要な問題は，クライアントがファイルに対してロックを要求し，そのあとクラッシュしてしまったらどうなるのかという点である．あるサーバーではロックされたままのファイルが残らないようにするために，ロックが行われると必ずタイマを起動し，ロックが解除される前にタイマが切れると，サーバーはクライアントがクラッシュしたものとみなし，ファイルのロックを解除する．しかしこの様な方法では，もしクライアントが単に低速で作業を行っているのみであり，異常がない場合には不都合が生じる．

同時実行制御は2章で学習した相互排除問題に非常に似ているが，若干異なっている．2章ではプログラムの側から問題を考え，危険領域を設定することによって，2つの危険領域が同時に使用されないようにした．ロックではデータの側からこの問題をとらえ，どのプログラムが実行中であるかに関係なく，各ファイルを直接保護していた．特定のファイルにアクセスする可能性のあるプログラムが多数存在する時，プログラムではなく，ファイルに制御を行ったほうがよいだろう．

実現方法ももちろん異なっている．なぜなら，ファイルシステム内にそれぞれのファイルに対するセマフォを持ち(まれにしか発生しない)，ロックに備えると，無駄が多くなる．その代わりロックされたファイルの一覧がメモリに保存される．また，クラッシュ，時間切れ，などのこのほかの部分も異なっている．

5.3.4 トランザクション

自動ロッキングは，トランザクション(transaction)という形でアトミック更新と組み合わされることもある．トランザクションは，成功すれば最後まで実行され，失敗すればシステムにはなんの影響も与えないという特徴を持っている．トランザクションを実行するには，クライアントプロセスがメッセージ "BEGIN TRANSACTION" をファイルサーバーに対して送信しなければならない．そして必要な数のファイルを読み取る．作業が完了すると，プロセスはメッセージ "END TRANSACTION" をファイルサーバーに送り，すべての変更内容をコミット済み(commited)，すなわち1つのアトミック更新において不変のものとする．コミットが不可能な場合は，トランザクションは失敗に終わり，なんの変化も起こらない．メッセージ "END TRANSACTION" を受け取るまでは，ファイルを使用している他のユーザーが書き込まれた内容を見ることはできない．他のユーザーが参照できるのは，修正前のファイルだけである．

複数のプロセスが同時にトランザクションを実行している時，ファイルサーバーはその時点で実行中であるのはそのプロセスのみであるという錯覚を起こさせる．つまり，ファイルサーバーは自動的にすべてのトランザクションを直列化しなければならない．ファイルサーバーが，直列化とアトミック更新を複数のファイルに対して効果的に行うための方法を見つけるのは，今後の課題である．ある簡単な方式を大まかに示すと，次の様になる．

プロセスがトランザクションを開始すると，ファイルサーバーは安定記憶機構上にトランザクション・レコード(transaction record)を作成し，その状態を記録する．安全記憶機構上に格納されているため，サーバー，ディスクのいずれがクラッシュしても失われることはない．プロセスが初めてファイルを読み取る際，ファイルサーバーはファイルをロックして他のプロセスのアクセスを防止する．ロックを行うことができなければ，トランザクションが失敗に終わり，何も変化は起こらない．

プロセスが初めてファイル上にブロックを書き込もうとすると，ファイルはロックされ，ファイルのコピーがとられる．書込みはオリジナルでなく，このコピーに対して行われる(そしてそのファイルに対する以降の書込みもすべてコピーに対して行われる)．

メッセージ "END TRANSACTION" が実行されると，サーバーは新しく作成されたファイルの集合と，まだ修正されていない古いファイル集合を置き換える．そしてインテンション・リスト(intentions list)を作成し，更新が必要なファイルと，それぞれの新しいファイル位置を表示する．インテンション・リストは安全記憶機構のトランザクション・レコードに加えられ，サーバーがクラッシュしても，再度立ち上げた時点で復旧できるようにする．そしてトランザクションが完了し，コミット済みと印付けするまで，他の操作は行えないことになる．

次に上書きされるファイル内のブロックをすべてについて，そのiノードをアトミック更新することにより，新しいファイルと置換する．この時点でトランザクションは完了し，すべてのロックが解除される．これらの手順を図5.18にまとめている．

クライアント	サーバー
トランザクション開始	トランザクションレコードの作成
ファイルAを読む	ファイルAをロック
ファイルBを読む	ファイルBをロック
ファイルAを更新	ファイルAのコピーと更新ファイルのコピーを作成
ファイルBを更新	ファイルBのコピーと更新ファイルのコピーを作成
トランザクション終了	安全記憶機構にインテンションリストを書き込む トランザクションをコミット済みと印付けする コピーによってファイルAを置き換える コピーによってファイルBを置き換える ロックを解除 確認する

図5.18 トランザクションの手順
(a)トランザクション中にクライアントが行う手順
(b)各手順においてサーバーが行う作業

これ以外にもより効率の良いアルゴリズムが発表されている．詳細はSvobodova(1984)の論文と，その参考文献を参照されたい．

5.3.5 複製ファイル

ファイルサーバーによっては，複製(replication)と呼ぶ便利な機能を用いているものがある．各ファイルを1つだけ保存するのではなく，サーバーはn個のファイルを保存する．コピーの1つが誤って破壊されても，データが失われることは決してない．標準ディレクトリが(ASCII名，iノード番号)のペアとしてリストされると，ディレクトリを，n個のファイルの(ASCII名，n個のiノード番号)を含むように変更し，複製を行うことができる．これを示したのが図5.19である．複製ファイルの詳細はPu他(1986)による文献などを参照されたい．

ASCII名	i-node
ファイル1	17
ファイル2	22
ファイル3	12
ファイル4	84

(a)

ASCII名	i-nodes		
ファイル1	17	19	40
ファイル2	22	72	91
ファイル3	12	30	29
ファイル4	84	15	66

(b)

図5.19 複製ファイル
(a)通常のディレクトリ (b)複製ファイルを持つディレクトリ

ファイルの1つのコピーが修正された時，その複製に対する操作はファイルサーバーが行う．2つの方法が考えられる．最初の方法はディレクトリにリストされている複製に対して，変更したブロックをそれぞれに送り，更新を行う．もう1つの方法はすでに古くなってしまった複製ファイルを削除して，修正済みのファイルの新しいコピーを作成し，これらをディレクトリに配置するという方法である．

異なるファイルサーバー上に複製ファイルが保存され，そのうちいくつかが複製作業中のネットワークのダウンにより，孤立してしまっても，分散されている他のコピーがバックアップ時に用いられることになる．複製ファイルがたまたまディレクトリであり，ネットワークがダウンしている間にすべてのコピーに対して個別に修正が行われた場合，ネットワークが再度立ち上がった時にファイルサーバーは一貫性を失った複数のディレクトリ:コピーを持つことになる．そこで，一貫性を取り戻すためになんらかの策を講じなくてはならない．この問題の詳細とその解決策についてはPopek他(1981)，Walker他(1983)，Weinstein他(1985)などの論文を参照されたい．

5.4 セキュリティ

ファイルシステムには，ユーザーにとって非常に重要な情報が含まれていることが多い．したがって，この情報を不正なユーザーから保護することはファイルシステムにとって最も大切なことである．以下の項ではセキュリティと保護に関する各種の問題について見ていくことにしよう．

5.4.1 セキュリティ環境

"セキュリティ"と"保護"という言葉は，同じ意味を持つ言葉として使われることも多いが，次の様な2つの一般的な問題として区別されている．ファイルが不正なユーザーによって読み書きされてはならないということから，技術的，管理上，法的，および政治的な要因も含まれている．もう1つはセキュリティを提供するための特定のオペレーティング・システムの機構である．本書では混乱を避けるために，全体的な問題を指す時にはセキュリティ(security)という言葉を，そしてコンピュータ内の情報を守るために用いられる特定のオペレーティング・システムの機構を指す時には保護機構(protection mechanisms)という言葉を用いることにする．ただしこの2つの境界はあまりはっきりと定義されていない．最初にセキュリティについて説明し，本章の後半において保護機構を見ていくことにしよう．

セキュリティには，多くの側面がある．そのうち最も重要なものを2つ挙げるとすれば，データの破壊と侵入であろう．データ破壊の主な原因としては，次の様なものが挙げられる．

- 天災：火災，洪水，地震，戦争，暴動，ネズミがカード，テープ，フロッピーディスクをかじったなど．
- ハードウェアまたはソフトウェア上のエラー：CPUの誤動作，読取り不能なディスクまたはテープ，通信エラー，プログラムのバグ．
- 人為的エラー：データの誤入力，誤ったテープやディスクのマウント，誤ったプログラムの実行，ディスクまたはテープの紛失．

このうち大部分はオリジナル・データからできるだけ離れたところに，適切なバックアップを保存することによって回避できる．

より興味深いのは，侵入問題の対処方法である．侵入には2つの形態がある．受動的な侵入は，読取り許可を与えられていないファイルの読取りである．能動的な侵入は，より悪意が強いものである．不正な変更を意図的に行おうとする場合がこれに相当する．システム設計において侵入に対するシステムの保護をより強力にするためには，どの様な侵入に対して保護を行うべきかを理解する必要がある．一般的な侵入方法には次の様なものがある．

●技術者以外の覗き．机上にタイムシェアリング・システムの端末を持っているユーザーは多く，詮索好きなユーザーの中には，特に障害がなければ他のユーザーの電子メイルやファイルの内容を見ようとする者もあるだろう．例えば大半のUNIXシステムでは特に制限を与えない限り，すべてのファイルを誰でも読み取ることができる．

●内部の人間による覗き．学生，システムプログラマ，オペレータ，あるいはその他の技術者によっては，コンピュータ・システムのセキュリティを破ることに意欲を燃している者もある．高度な技術を持ち，かなりの時間を費やして目的を達しようとすることもある．

●一儲けしようという人．銀行からお金を盗むために銀行システムに侵入しようとした，銀行のプログラマがいた．ソフトウェアに修正を加え，四捨五入ではなく，切り捨てを行わせたり，セント以下を切り捨てることによって利益を得たり，何年間も使用されていない口座から預金を吸収したり，脅迫状を送ったりと("要求に応じなければ，銀行の取引記録をすべて破壊してしまう"などと脅す)，その方法は様々である．

●産業または軍事スパイ．競合企業または外国など資金力のある団体が行うスパイ行為．プログラム，企業機密，特許，技術，回路設計，営業プランなどを盗もうとする行為である．この様な行為は，しばしば盗聴や，コンピュータからの電磁放射を捕えるためのアンテナを使ったりする．

KGBから軍事機密を守ろうとする場合と，学生が奇妙なメッセージをマシンに不正に挿入するのを防ぐ場合では事の重大さが違う．セキュリティと保護にかける意気込みは，誰が敵であるかによって明らかに違ってくる．

セキュリティのもう1つの側面は，プライバシー(privacy)である．すなわち個人情報の誤用を防ぐことが必要である．これは多くの法的および道義的問題を引き起こすことになる．もし政府が独自のやり方で各個人の身上書を作成し，社会保険の費用や税金をごまかした人を洗いだそうとしたらどうなるだろう．警察は組織犯罪を防止するために，誰のどの様な情報でも見ることができるのだろうか．雇用者や保険会社はその様な権利を持っているだろうか．これらの権利と個人の権利が矛盾する時はどうすればよいのだろうか．これらの問題はすべて非常に重要であるが，本書では言及しないことにする(参考文献に関してはAPPENDIX Eを参照されたい)．

5.4.2 セキュリティの落とし穴

運輸業界でのタイタニックやヒンデンブルグの様に，コンピュータの機密保護の専門家にも思い出したくない事がいくつかある．この項では，4つのオペレーティング・システム(UNIX, MULTICS, TENEX, およびOS/360)で発生したいくつかのセキュリティの問題を見ていくことにする．

UNIXのユーティリティであるlprは，ラインプリンタにファイルを印刷するものである．これ

には印刷後ファイルを削除するというオプションが提供されている．UNIXの初期バージョンでは，誰もがlprを使ってパスワード・ファイルを印刷し，その後システムにパスワード・ファイルを削除させることができた．

UNIXに侵入するもう1つの方法は，ワーキングディレクトリ内に存在するcoreという名前のファイルをパスワード・ファイルにリンクするものである．そして侵入者がSETUIDを持ったプログラムを実行した際，コアダンプを強制的に引き起こすことにより，システムはパスワード・ファイルの先頭からcoreファイルを書き込む．こうしてユーザーはパスワード・ファイルを，自分の選んだいくつかの文字列(例えばコマンド引数など)を持つファイルに置き換えることができる．

さらにUNIXには以下のコマンドに関わる問題もある．

```
mkdir  foo
```

mkdirは，rootが所有者であり，SETUIDを持つプログラムである．システムコールMKNODによってディレクトリfooのiノードを作成後，fooの所有者を有効uid(すなわちroot)から，実uid(ユーザーのuid)に変更する．システムが低速で動作している時，ユーザーはディレクトリiノードを素早く削除し，MKNODとCHOWNの間にfooという名前でパスワード・ファイルに対するリンクを作成できることがある．mkdirがCHOWNを実行すると，そのユーザーがパスワード・ファイルの所有者になる．必要なコマンドをシェルスクリプトとして用意することで，トリックが成功するまで何度もやり直すことができる．

MULTICSにおけるセキュリティの問題は，システム設計者がMULTICSをタイムシェアリング・システムとして考えており，頑固なバッチ信仰者のために後から思い付きでバッチ機能を追加したという点に起因している．タイムシェアリングのセキュリティは優れていた．しかしその一方ではバッチシステムにおけるセキュリティが存在すらしていなかった．任意のユーザーディレクトリに多数のカードを読み出させるようなバッチジョブを挿入することぐらいであれば，誰にでも行うことができた．

誰かのファイルを盗むためには，エディタのソースコードを入手し，ファイルを盗むよう修正し(ただしエディタの機能はそのまま)，それを犠牲者のディレクトリbinに置いておく．その後犠牲者がエディタを呼び出すと，侵入者の作成したエディタが実行される．エディタは普通に動くが，犠牲者のファイルが盗まれてしまう．通常のプログラムを修正し，通常の機能以外にも悪意の機能を持たせ，犠牲者に修正したものを使用させるという技法はトロイの木馬(Trojan horse attack)と呼ばれている．

TENEXオペレーティング・システムは，かつてDEC-10コンピュータ上で，人気を博していた．今ではそれほど人気はなくなったが，次の様な設計上の誤りにより，コンピュータ・セキュリティの歴史に永久に残ることになってしまった．TENEXもページングをサポートしていた．ユーザーは自身のプログラムの動作を監視するために，ページフォルト発生のたびにユーザー関

数を呼び出すようシステムに指示することができた.

TENEX ではパスワードを用いてファイルを保護することも可能であった. ファイルにアクセスするために, プログラムは正しいパスワードを提示しなければならない. オペレーティング・システムは1文字ずつパスワードを検査し, 誤りを発見した場合には, ただちに停止する. TENEX に侵入するためには, 図5.20(a)に示すように, 最初の文字を1ページの最後に配置し, 残りを次のページの最初に置くといった操作を注意深く行う.

次の手順は, 例えば非常に多くのページを参照することにより, 2ページ目をメモリから追い出し, 2ページ目をメモリ内に置かないようにすることである. ここでプログラムは用意されたパスワードを用いて, 犠牲者のファイルをオープンしようとする. 実際のパスワードの最初の文字がA以外ならば, システムは最初の文字の検査で停止し, ILLEGAL PASSWORD を返す. しかし実際のパスワードがAから始まる場合, システムは検査を続け, ページフォルトに遭遇した時点で侵入者に通知される.

パスワードがAで始まらない場合は, 侵入者は図5.20(b)の様なパスワードに変更し, プロセス全体を繰り返して, Bから開始するかを調べる. 最高128回このプロセスを繰り返して, ASCII 文字のすべてを調べれば, 最初の文字が判明する.

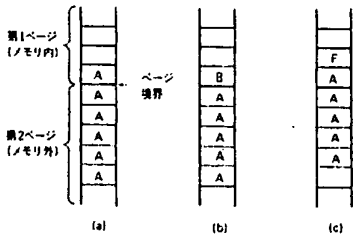

図5.20 TENEX パスワードの問題点

最初の文字がFだったとしよう. 図5.20(c)のメモリレイアウトで侵入者はFA, FB といった文字列の検査を行うことができる. この方法を用いることにより, n 文字のパスワードを見つけ出すために必要な試行回数は, 128ⁿではなく, 128×n 回となる.

最後の問題は OS/360 に関するものである. 以下は概略であるが, 問題の基本的な部分は解説している. このシステムではテープの読取りを開始し, ドライブがデータをユーザー空間に転送している間も, 演算を行うことが可能であった. そのトリックとは, まず注意深くテープの読取りを開始し, 次にユーザーのデータ構造体(例えば読み取るファイル名とそのパスワード)を必要とするシステムコールを発行することである.

このオペレーティング・システムは, 最初にパスワードが指定したファイルに対して正しいもの

であるかを確認する. 次に, 実際にアクセスを行うために再度ファイル名を取り出そうとする(ファイル名をシステム内部に保存しておけばよかったのだが, 実際には行っていなかった). 残念ながら, システムが2回目にファイル名を取り出す直前に, ファイル名はテープドライブからのデータによって上書きされている. そしてシステムはパスワードの検査を行わずに新しいファイルを読み出してしまう. この様にタイミングをうまくつかむにはある程度経験が必要だが, それほど難しいことではなかった. しかもコンピュータが得意とする, 同一操作の無数の繰り返しが含まれていた.

5.4.3 一般的なセキュリティへの攻撃

前述の様な問題はすでに改善されてはいるが, 平均的なオペレーティング・システムではまるでふるいの様に情報が漏れているのが実状である. システムのセキュリティ状態を知る一般的な方法としては, タイガーチーム(tiger teams)や, 透過チーム(penetration teams)と呼ばれる専門家グループを雇い, 侵入が可能かを調べてもらう. Hebbard(他 1980)も同じことを大学院生を使って行った. 透過チームは何年かの期間をかけ, システムの弱点と考えられる部分をいくつか発見した. 以下に成功する確立の高い攻撃方法をいくつか挙げてみた. システムを設計する際, これらの点には特に注意されたい.

- メモリページ, ディスク空間, テープを要求し, それを読み取る. 多くのシステムは割当て前にそれを消去することはなく, 前の所有者が書き込んだ興味深い情報で満たされているかも知れない.

- 不正なシステムコール, 不正な引数を持つシステムコール, または不正ではないが意味のない引数を持つシステムコールの使用. 多くのシステムは混乱を生じる.

- ログインを開始し, DEL, RUBOUT, BREAK をログイン・シーケンスの途中で入力する. システムによってはパスワード検査プログラムが強制終了させられ, ログインが成功したとみなされる.

- ユーザー空間に保存されている複雑なオペレーティング・システムの構造体を修正してみる. 多くのシステムではファイルをオープンするために, プログラムが大きなデータ構造を用意し, それにファイル名やその他多くの引数を設定し, システムに渡す. ファイルの読み書きを行うと, システムは構造体自身も更新することがある. これらのフィールドを変更するとセキュリティに大きな障害が出てくる.

- 画面に "login:" を表示するプログラムを作成し, ユーザーを混乱させる. ユーザーの多くは端末に向かってログイン名と, パスワードを入力してしまい, プログラムはそれを侵入のために記録する.

- マニュアルで"~してはならない"と記されている事項を探し，できるだけ多くの禁止事項を行ってみる．
- システムプログラマを説得し，特定のログイン名を持つユーザーに対して重要なセキュリティ検査のいくつかをとばすようシステム内容を変更させる．この攻撃方法は落とし穴(trapdoor)技法として知られている．
- 上述のいかなる方法も使用することができない時は，コンピュータ・センタの重役秘書を買収する．秘書はたいてい貴重な情報に簡単にアクセスすることができ，しかも安い賃金で働いているはずだ．人的要因の効果をあなどってはいけない．

これ以外にも各種の攻撃方法がLinde(1975 NCC)によって紹介されている．

5.4.4 セキュリティの設計理念

SaltzerおよびSchroeder(1975)は，安全なシステムを設計するためのいくつかの基本理念を発表した．両者の概念を以下に簡単にまとめる(MULTICSでの経験に基づいている)．

第1に，システムの設計技法を公開する．侵入者がシステムの仕組みを知らないだろうと，過信してはならない．

第2に，デフォルト値をアクセス不可にする．合法的アクセスが拒絶されるようなエラーは，不正アクセスが許される前にいち早く報告する．

第3に現在与えられている権利を調べる．システムは，許可を調べ，アクセスを許すかどうかを決定するが，その情報を後で使用するために保存してはならない．多くのシステムはファイルをオープンする際に許可内容を調べても，以後確認は行っていない．したがってユーザーがファイルをオープンしたまま何週間も経過すると，すでにファイルの所有者がファイルの保護内容を変更してかなり経過していても，ユーザーは変更前の保護内容のままでそのファイルに引き続きアクセスすることができる．

第4に各プロセスに最低限の特権しか与えないようにする．エディタが特定のファイル(エディタを呼び出す時に指定した)を編集するためのアクセス権しか持っていないとしたら，トロイの木馬の手法を用いたエディタもあまり効果はない．この理念はきめ細かい保護手法を示している．この章の後半でその様な手法に関して説明する．

第5に保護機構は単純で，一貫性を持っており，システムの最下位に組み込まれていなければならない．保護機能の弱い既存システムに合わせてシステムを改造することはできない．保護機能は，正当性と同じで後から追加できる機能ではない．

第6に選んだ手法は心理的に受け入れやすいものでなくてはならない．ユーザーがファイルの保護を面倒だと感じるなら，保護は行わないだろう．しかしいったん問題が発生すると，不満を爆発させることが多い．"それはあなたのせいです"といってもそれは受け入れられない．

5.4.5 ユーザーの認証

多くの保護手法は，システムがそれぞれのユーザーを識別できるという仮定に基づいている．ユーザーがログインした時に行われる識別は，認証(user authentication)と呼ばれている．多くの認証方法はユーザーの知っていること，持っているもの，ユーザー自身などに基づいている．

■ パスワード

認証の最も一般的な形態は，ユーザーにパスワードを入力させることである．パスワードを使った保護はわかりやすく，実行も簡単である．UNIXでは以下の様な働きをする．ログイン・プロセスがユーザーに名前とパスワードを入力するよう促す．パスワードは直ちに暗号化される．次にログイン・プログラムは1ユーザーあたり1行のASCII文字列から構成されるパスワード・ファイルを読み取り，ユーザーのログイン名を含む行を検出する．この行に含まれたパスワード(暗号化された)が，入力したものを暗号化したパスワードと一致すれば，ログインは認められ，そうでなければログインは失敗に終わる．

パスワードによる認証を破るのも簡単である．事実，大企業や政府機関の所有する極秘システムに家庭用コンピュータを使って侵入した高校生や，中学生グループに関する記事を頻繁に見かける．本質的には侵入は，いつでもユーザー名とパスワードの組み合わせを予測することによって行われる．

MorrisおよびThompson(1979)は，UNIXシステムにおけるパスワードの研究を行った．彼らは一般的なパスワードをリストにまとめた．リストによると，姓名，氏名，通りの名前，町名，普通の辞書に収録されている言葉(そしてそれを逆に綴ったもの)，車のナンバープレートの番号，ランダムな文字から成る短い文字列などが用いられやすいという．

そして次にこれらを1つずつ既知のパスワード暗号化アルゴリズムを用いて，暗号化した．そして暗号化されたパスワードのうちリストのエントリと一致するものがないか調べたところ，パスワードのうち86%がリストに記載のものであったという．

印字可能な95のASCII文字の中からランダムに抽出された7文字だけをパスワードに使用する場合は，検索空間は95^7，すなわちおよそ7×10^{13}となる．1秒あたり1,000回暗号化作業を行う場合，パスワード・ファイルとの照合を行うためのリストを作成するには，2,000年もかかることになる．さらにリストは磁気テープ20,000,000本もの長さになる．パスワードに小文字と，大文字，そして特殊文字をそれぞれ少なくとも1つ以上含ませ，7文字以上にすることだけでもユーザーが選択するパスワードはかなりの改善をもたらす．

ユーザーに適切なパスワードを選択させることが実質的に不可能だとしても，MorrisとThompsonはその様な攻撃(あらかじめ多数のパスワードを暗号化しておく)を無意味にする技法を紹介している．彼らのアイデアはnビットのランダムな番号を，各パスワードに連結するというものである．パスワードが変更されればランダムな番号も変更される．ランダムな番号は暗

号化される前の形でパスワード・ファイルに保存されており，誰にでも読み出せるようになっている．暗号化されたパスワードをパスワード・ファイルに保存するため，パスワードとランダムな番号が最初に連結され，共に暗号化される．暗号化されたものはパスワード・ファイルに保存される．

ここでパスワードに使われそうなもののリストを作って暗号化し，その結果を分類済みのファイルfに保存し，暗号化された任意のパスワードを簡単に検索しようとしている侵入者を考えてみよう．侵入者がパスワードがMarilynであると予想したら，Marilynを暗号化してその結果をfに入れるだけでは不十分である．Marilyn0000, Marilyn0001, Marilyn0002など，2^n分の暗号化を行い，すべてをfに保存しなくてはならない．この技法によってfの大きさも2^nになる．UNIXはこの技法をn=12で使用している．

この技法はあらかじめ暗号化された多数のパスワード・リストを作成することによって侵入をたくらむ者に対しては保護効果があるが，Davidというユーザーが，そのパスワードもDavidとしている場合には，なんの効果もない．より適切なパスワードを選択させるためには，コンピュータになんらかの提案を行わせるとよい．コンピュータによってはfotally, garbungyまたはhipittyなどパスワードとして使用できる，発音の簡単な，しかも意味をなさない音節(できるだけ大文字と特殊文字が盛り込まれているもの)を作成するプログラムを持っているものもある．

また別のコンピュータではユーザーにパスワードを定期的に変更させ，パスワードが誰かに知られた場合も，被害を最小にとどめている．この技法の極端な例は一時パスワード(one time password)である．一時パスワードが使用される場合，パスワードのリストを含んだ冊子を渡されることになる．ログインのたびにリスト内の次のパスワードを用いる．侵入者がもしパスワードを発見しても，次回から異なるパスワードを使用することになるので，どうにもならない．ユーザーはこの冊子をなくさないように心がけなくてはならない．

いうまでもなく，パスワードが入力されてもコンピュータは入力された文字を表示してはならない．端末の近くに悪意のユーザーがいる可能性があるからである．またパスワードを暗号化される前の形態でコンピュータに保存しないこと，そしてコンピュータ・センタの管理者にも暗号化される前の形態で保存しないことなども重要である．暗号化される前のパスワードがどこに保存されても，問題になりやすい．

また新しいユーザーに長い質疑応答リストを提供させ，暗号化された形態でそれをコンピュータに保存するといったパスワード設定技法もある．質問内容はユーザーが書き留めなくても選択できるような形式のものである．典型的な質問例を以下に挙げる．

- Marjoleinの妹の名は？
- 小学校が位置していた通りの名前は？
- Mrs. Woroboffが教えている教科は？

ログイン時にコンピュータはランダムに1つの質問を抽出し，その答を確認する．

もう1つの方法はチャレンジ・レスポンス(challenge-response)と呼ばれている．この技法を用

いると，ユーザーはユーザー登録中に例えばアルゴリズムとしてx^2を選択する．そしてユーザーがログインすると，コンピュータは引数として例えば7を表示し，その場合ユーザーは49と入力する．アルゴリズムは週の曜日や，端末，また朝と夜でも異なっている可能性がある．

■ 物理的識別

全く異なった認証方法として，ユーザーがなんらか，例えば磁気ストライプを貼っているプラスチック・カードを持っているかどうかを確認するというやり方もある．カードを端末に挿入させ，そこで誰のカードであるかを判別する．この方法をパスワードと組み合わせて，(1)カードを持っており，(2)パスワードを知っているユーザーのみログインを許可することもできる．自動預金引出機がこの典型的な例である．

さらにもう1つの方法は，模造の困難な物理的特徴を測定することである．例えば端末に指紋または声紋鑑定機を付けてユーザーを識別することができる(ユーザーがコンピュータに自分が誰であるかを告げ，特定の指紋と照合すれば，データベース全体を照合するよりも検索は速く行える)．直接目で見て判断することはまだできないが，今後その様な機能が提供される可能性もある．

そしてもう1つのアプローチとして署名を用いるものもある．ユーザーは端末に接続された特殊なペンで署名し，コンピュータがそれを登録されている署名と比較する．署名の上手下手ではなく，署名中のペンの動きを比較する．署名を真似ることはできても，どのストロークを先に打つかまで知ることはできない．

これ以外にも指の長さを分析するという方法があるが，これは非常に効果的である．この技法を用いるためには，各端末に図5.21の様な装置を用意する．ユーザーは手をそこに挿入し，すべての指の長さが測定され，データベースと照合される．

さらに例をあげ続けることもできるが，以下の2つを重要例として紹介する．ネコやその他の動物は自分の領域の周りに排泄することによって，その境界を守ろうとする．例えば検尿用の器械を用いて非常に簡単な識別を行ったとしよう．各端末にはこの器械が1台備わっており，それぞれ"ログインの際はサンプルをここに入れてください"という内容のメッセージが現れる．これは絶対侵入することのできないシステムだが，おそらくかなり深刻なユーザー側の受け入れに関する問題が発生するだろう．

同様なことが画鋲と小型の分光器から成るシステムにもいえる．ユーザーは画鋲で親指を刺すよう指示され，それによって出た血を分光器による分析にかける．重要な点は，どの様な認証方法であれユーザー社会において精神的に受け入れられなければならないということである．指の長さを測る程度なら特に問題はないが，コンピュータに指紋を保存しておくことすら受け入れたくない人も多い．

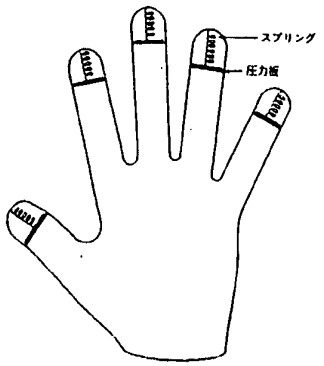

図5.21 指の長さを測定する装置

■ 対処方法

侵入者がシステムに入り込んで，多大な被害を引き起こした後，厳重な保護機構を採用することになったシステムでは，不正な入力が非常に困難になるような措置を取ることが多い．例えば各ユーザーは特定の端末からしかログインすることができず，しかもログインできるのは特定の曜日と，特定の時間といった制限付きであることも多い．

ダイヤル回線を以下の様に使用するのも1つの方法であろう．だれでもダイヤルを回して，ログインすることはできるが，ログインが成功すると，システムは直ちに回線を切り，指定された番号のユーザーを呼び返す(コールバック)．これによって侵入者は任意のダイヤル回線から侵入することができなくなり，ユーザーの(特定の)電話だけにその権利が与えられることになる．コールバックを行う場合も，行わない場合もシステムはダイヤル回線から入力されたパスワードを検査するために少なくとも10秒を要する．ただしこの時間は何度か連続してログインに失敗すると延長され，侵入者の侵入率を低下させることができる．また，ログインに3回失敗すると，回線は10秒間断線され，その間に保護管理者に通報するというのも効果的である．

ログインの記録はすべて保存しておかなければならない．ユーザーがログインすると，システムは前回のログイン時間と端末を報告し，それにより侵入を知ることができる．

次のステップとして侵入者を捕えるための罠を仕掛けなければならない．簡単なパスワードを持つ特殊なログイン名(例えばログイン名：guest，パスワード：guest)を提供する方法が簡単である．この名を用いてログインしたものがあればシステム保護管理者に通報される．これ以外の方法として，オペレーティング・システムに発見しやすいバグを仕掛けたり，侵入者の捕獲を目的とした同様の仕掛けを行うことである．

5.5 保護機構

ここまでの節では，多くの問題点について見てきた．その中には，技術的なものもあれば，そうでないものもあった．この節ではファイルその他の保護を行うためにオペレーティング・システムで用いられているさまざまな技法を解説していきたい．これら技法はすべて方針(どのデータを誰から守るのか)および機構(方針に従うためにどの様なシステム形態をとるか)の区別が明確になされている．方針と機構の区別に関しては Levin 他(1975)を参照されたい．ここでは方針ではなく，機構に重点をおいて説明していく．

5.5.1 保護ドメイン

コンピュータ・システムには保護を必要とする対象物(objects)が多数含まれている．対象物は，CPU，メモリセグメント，端末，ディスクドライブ，プリンタなど，ハードウェアである場合もあれば，プロセス，ファイル，データベース，セマフォといったソフトウェアである場合もある．

各対象物には，参照に用いられる一意な名前が付けられており，また対象物に対して実行可能な一連の操作を持っている．ファイルに対しては READ および WRITE が，またセマフォには UP と DOWN が適切である．オペレーティング・システムにおける対象物は，プログラミング言語での，抽象データ型(abstract data types)と呼ばれるものに相当する．

いうまでもなく，プロセスがアクセス権を与えられていない対象物にアクセスできないようにするための方法が必要である．さらにこの機構は，必要に応じてプロセスによる特定の正当な操作も禁止できなくてはならない．例えばプロセスAはファイルFの読取りは行えても，書込みはできない，といった指定も可能である．

異なる保護機構について述べるために，ドメインに関する説明を加えておくことにする．ドメイン(domain)は(対象物，権利)のペアの集合である．各ペアは対象物とそれに対して行うことのできる操作の部分集合を構成している．ここでの権利(rights)とは，これらの操作のいずれかを実行するために与えられる権利を指す．

図5.22には3つのドメインを示しており，各ドメインには対象物とその権利(Read，Write，eXecute)が含まれている．プリンタ1が同時に2つのドメインに存在している点に注意されたい．この例では示していないが，同じ対象物を複数のドメインに置き，各ドメインに異なる権利を与えることも可能である．

プロセスは，常になんらかの保護ドメイン内で実行されている．すなわちアクセス可能な対象物の集まりが存在し，各対象物に対してなんらかの権利を与えられている．プロセスは実行中に1つのドメインから別のドメインに移動することも可能である．ドメインの切り換えに関する規定はシステムによって大きく異なっている．

保護ドメインの概念をさらに明らかにするため，UNIX を見てみよう．UNIX ではプロセスの

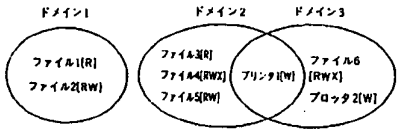

図5.22 3つの保護ドメイン

ドメインはuidとgidで決定される．どの様な(uid, gid)の組み合わせを用いても，アクセス可能な対象物(特殊ファイルとして表現されている入出力装置も含めたファイル)をすべてリスト表示したり，読み書き実行のいずれに対するアクセス権を与えられているのかを知ることができる．同じ(uid, gid)の組み合わせを持つ2つのプロセスは，同一の対象物の集合にアクセスできる．異なる(uid, gid)値を持つプロセスは，異なるファイルにアクセスを行うことになるが，たいていかなりの量のファイルが重複している．

さらにUNIXのプロセスはユーザー部分とカーネル部分を半分ずつ持っており(3章の図3.14)，カーネル部分はユーザー部分と異なる対象物のセットにアクセスを行うことができる．例えばカーネルは物理メモリ内のページ，ディスク全体，その他保護資源すべてに対してアクセスすることができる．したがってシステムコールはドメインの切り換えを引き起こす．

プロセスがSETUIDまたはSETGIDビットを持つファイルをEXECにより実行した場合，新しい有効uidまたはgidを得る．異なる(uid, gid)の組合わせを用いると，異なるファイルと操作セットを指定することができる．SETUIDまたはSETGIDを持つプログラムの実行もドメインを切り換えることになる．

UNIXでは，プロセスをカーネル部分とユーザー部分に分割しているが，これはMULTICSで用いられていた，より強力なドメイン切り換え機構を受け継いだものである．MULTICSでは，ハードウェアがサポートしていたプロセスあたりのドメインは2つ(カーネルとユーザー)ではなく，最大64であった．MULTICSのプロセスはそれぞれがなんらかのドメインで実行中の手続きから構成されていた．ここでのドメインはリング(rings)と呼ばれていた(Schroeder, Saltzer, 1972)．手続きは実行中のプロセスに動的にリンクされることができる．

図5.23は4つのリングを示したものである．最も内側のリングはオペレーティング・システムのカーネルであり，最も強力である．カーネルから外に移動するにつれ，リングの力は弱くなる．例えばリング1にはUNIXにおけるmkdirなどの様な，rootが所有し，SETUIDを持つプログラムによって処理される機能が含まれている．リング2は学生のプログラムを評価するための採点プログラムが，そしてリング3には学生プログラムが含まれている．

1つのリング内の手続きが別なリングの手続きに呼び出されると，トラップが発生し，システムはプロセスの保護ドメインを変更することができる．したがってMULTICSプロセスは，一生のうちに最大64の異なるドメインで実行されることができた(実際には上述の状況よりも複雑なこ

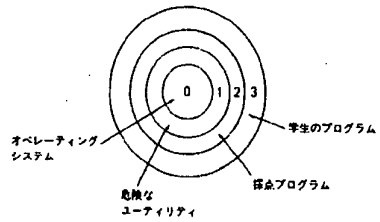

図5.23 4つのリングを用いたMULTICSのプロセス
各リングは異なる保護ドメインに存在する．

とが多い．手続きは複数の隣接するリングで実行することが可能で，リング間でやりとりされる引数は入念に制御されていた)．MULTICSに関する詳細は，Organick(1972)を参照されたい．

それでは，システムはどの様な方法でどの対象物がどのドメインに属しているかを記録しているのだろうか．概念的には少なくともドメインを行とし，対象物を列とする大規模なマトリックスを想像することができる．図5.22のマトリックスを示したのが図5.24である．このマトリックスと現在のドメイン番号を用いて，システムは常に指定したドメインから特定の方法で特定の対象物に対するアクセスが可能かを判断できる．

ドメイン＼対象物	ファイル1	ファイル2	ファイル3	ファイル4	ファイル5	ファイル6	プリンタ1	ブロック2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

図5.24 保護マトリックス

ドメインの切り換え自体は，MULTICSと同様，ドメイン自身が対象物であるとみなし，ENTERの操作を行うことによって，マトリックス・モデルに容易に含ませることができる．図5.25は図5.24のマトリックスを再度図示したもので，今回は3つのドメインを対象物自身として表している．ドメイン1のプロセスはドメイン2に移行することができるが，移行した後はもうもとに戻ることができない．この様な状態は，UNIXにおけるSETUIDを持つプログラムの実行を表したものである．この例ではこれ以外のドメイン切り換えは認められない．

対象物

ドメイン	ファイル1	ファイル2	ファイル3	ファイル4	ファイル5	ファイル6	プリンタ1	プロッタ2	ドメイン1	ドメイン2	ドメイン3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

図5.25 対象物としてのドメインを持つ保護マトリックス

5.5.2 アクセス制御リスト

実際には図5.25のマトリックスは非常に大きく，分散しているため，これを保存することは滅多にない．大半のドメインはほとんどの対象物に対して全くアクセス権を与えられていないため，大きくて，中身のないマトリックスを保存することはディスク空間の浪費につながる．それでもマトリックスの保存が必要な場合は，行または列単位で，しかも空きエレメントは除いて保存するようにする．2つのアプローチは驚くほど異なっているため，この項では列単位での保存について触れ，次の項において行単位での保存について触れることにする．

最初の技法では，各対象物にリスト(一定の順番を付けた)を連結させる．このリストには対象物にアクセスすることのできるすべてのドメインと，その方法が記載されている．このリストはアクセス制御リスト(access control list)またはACLと呼ばれている．UNIXで実現する場合，最も簡単な方法は，ACLを各ファイルの個別ディスクブロックに置き，ファイルのiノードにブロック番号を記録することである．マトリックスのうち，空のものは対象となっていないため，すべてのACL保存容量はマトリックス全体に比べるとかなり少なくて済む．

ACLの仕組みを知るために，引き続きUNIXにおいて(uid, gid)ペアでドメインが表現される例を考えてみよう．事実ACLはUNIXの前身であるMULTICSで，以下の様な形態で使用されていた．したがって，ここで挙げる例は単なる空想上のものではない．

ここではJan, Els, Jelle, Maaikeという4人のユーザー(すなわちuid)があると仮定しよう．それぞれのユーザーはシステム，スタッフ，学生，学生というグループに属している．いくつかのファイルが以下のACLを持っているとしよう．

File0：(Jan, *, RWX)

File1：(Jan, system, RWX)

File2：(Jan, *, RW-), (Els, staff, R--), (Maaike, *, R--)

File3：(*, student, R--)

File4：(Jelle, *, ---), (*, student, R--)

カッコ内のACLエントリはuid, gid, および許可されているアクセス内容(Read, Write, eXecute)を示している．アスタリスクはすべてのuidまたはgidを表している．File0はuid=Janなら，任意のgidを持つプロセスによって読み書き実行を行うことができる．File1はuid=Janでgid=systemのプロセスだけがアクセス権を与えられている．uid=Janでgid=staffのプロセスはFile0にアクセスすることはできても，File1にアクセスすることはできない．File2はuid=Janで，任意のgidを持つプロセスか，uid=Elsで，gid=stallのプロセス，またはuid=Maaikeで，任意のgidを持つプロセスのいずれかによって読み書きが行える．File3は学生ならば誰でも読取ることができる．File4は特に興味深い．uid=Jelleならば，グループに関係なくアクセスを全く認めないというものである．ただし他の学生はそれを読取ることができる．ACLを用いると，同じクラスの他の者にはアクセスを与えながら，特定のuidまたはgidによる対象物のアクセスだけを禁じることができる．

この様な機能はUNIXにはない．それではUNIXに提供されているものを見てみることにしよう．UNIXではファイルの所有者，所有者のグループ，一般ユーザーそれぞれに対して3つのビットrwxを提供している．手法はACLと同じだが，9ビットに縮めてある．これは対象物に連結されたリストで，誰がどの様にしてその対象物にアクセスできるかを示している．9ビットを用いたUNIXのスキーマはACLシステムに比べてあまり一般性はないが，実際にはこの技法の方が優れており，実現も簡単でありコストも安い．

対象物の所有者はいつでもACLを変更し，それまで与えられていたアクセス権利を簡単に禁止することができる．唯一問題となるのは，ACLを変更しても現在その対象物を使用しているユーザー(例えばファイルをオープンしているなど)には，おそらくなんの影響もないという点である．

5.5.3 権 限

図5.25のマトリックスを行単位で切り分けることもできる．この方法を用いると，各プロセスにはアクセスすることのできる対象物のリストと，それぞれに認められている操作，すなわちドメインが連結される．このリストは権限リスト(capability list)と呼ばれており，そこに置かれた個々の項目を権限(capabilities)と呼んでいる(Dennis Van Horn, 1966；Fabry, 1974)．

典型的な権限リストを図5.26に示している．それぞれは対象物の種類を示すフィールドtypeと，この種の対象物に認められた正当な操作を表したビットマップであるフィールドrights，そして対象物自身を指すポインタであるフィールドobject(例えばiノード番号など)を持つ．権限リスト自体も対象物であるため，他の権限リストからポインタを使って指すことも可能である．これはサブドメインの共有を可能にする．許可は権限リストにおけるその位置によって参照されることが多い．プロセスが"権限2が指しているファイルから1Kを読み出せ"という要求を行ったとしよう．この様なアドレス付けの形態は，UNIXにおけるファイル記述子の使用と似ている．

権限リスト，すなわちCリスト(C-lists)は，ユーザーによる不正な変更から守らなくてはなら

	タイプ	Rights	対象物
0	ファイル	R--	ファイル3へのポインタ
1	ファイル	RWX	ファイル4へのポインタ
2	ファイル	RW-	ファイル5へのポインタ
3	プリンタ	-W-	プリンタ1へのポインタ

図5.26 図5.24におけるドメイン2の権限リスト

ない．そのために3つの方法が提供されている．最初の方法ではタグアーキテキチャ(tagged architecture)と呼ばれるものが必要である．このアーキテクチャでは，権限の有無を示す予備(またはタグ)ビットを各メモリワードに付加する．タグビットは演算，比較，その他通常の命令には使用されない．そしてカーネルモードで実行中のプログラム(すなわちオペレーティング・システム)だけがそれを変更することができる．

2番目の方法としてはオペレーティング・システムの内部にCリストを保存し，上記の様にスロット番号を使ってプロセスが参照を行うというものである．Hydra(Wulf, 1974)はこの様な構造を用いていた．3番目の方法としては，Cリストをユーザー空間に置きながら，各権限をユーザーの知らない秘密のキーで暗号化するのである．この方法は特に分散システムには効果的であり，Amoebaでも使用されている(Tanenbaum他, 1986)．

読出しや実行など対象物に依存する権利のほかに，すべての対象物に適用される一般的権利(generic rights)が権限に提供されている．一般的権利の一例を以下に挙げる．

- 権限コピー：同一対象物に新しい権限を作成する．
- 対象物・コピー：新しい権限を持つ複写対象物を作成する．
- 権限削除：対象物はそのままで，Cリストからエントリを削除する．
- 対象物破壊：対象物と権限を永久に消去する．

多くの権限システムは，対象物の各タイプに対してタイプマネージャ・モジュール(type manager module)を持つ，モジュール集合から構成されている．ファイルに対して操作実行を促す要求はファイルマネージャに，そしてメイルボックスに関する要求はメイルボックス・マネージャに送られる．これらの要求は該当する権限を伴っている．タイプマネージャ・モジュールは通常のプログラムであるため，ここで問題が生じる．ファイルの所有者はファイルの操作のいくつかを実行することができるが，その内部表現(例えばiノードなど)を得ることはできない．タイプマネージャ・モジュールに通常のプロセス以上の権限を持たせることが必要である．

Hydraでは権利の拡大(right amplification)という技法でこの問題を解決した．この技法ではタイプマネージャに特定の権利テンプレートを与え，このテンプレートによってそれ自身が認められているより多くの権利をオブジェクトに与えることができる．対象物を強く型付けし，分類している他の権限システムでも，同様の技法が用いられている．

権限システムに関して最後に触れておかなければならないのが，対象物に対するアクセスを無

効にすることがかなり困難であるという事実である．システムが任意の対象物に対して外部にある権限を検出し，操作することは困難である．ディスク上に分散されているCリストに保存されている可能性もあるからである．1つの解決策として各権限に対象物自身ではなく，間接対象物を指させるという方法がある．間接対象物に，実際の対象物を指させることにより，システムは常にその接続を断ち，権限を無効にすることができる(後から間接対象物に対する権限がシステムに提示されると，ユーザーは間接対象物がなにも対象物を指していないことを発見する)．

許可を無効にするためのもう1つの方法として，Amoebaで使用されている手法がある．各対象物は長いランダムな番号を含んでおり，これと同じものが権限にも提供される．権限を使用するために番号を提示すると，その2つの番号が比較される．番号が一致した場合に限り，操作が認められる．対象物の所有者は，対象物のランダムな番号の変更を要求し，既存の権限を無効にすることができる．これら手法のいずれも選択的な無効，つまりJohnには権限を与えるが，他者には与えないといった指定を行うことはできない．

5.5.4 保護モデル

図5.24の保護マトリックスは静的なものではない．新しい対象物の作成，古い対象物の破壊，そして所有者が対象物に対するユーザー数の増減を行う際に，頻繁に変更が行われる．保護システムの構築はかなり注意深く行われており，それにより保護マトリックスは絶えず変更されている．以下の項ではこの作業について簡単に触れることにする．

Harrisonら(1976)は，あらゆる保護システムの基本として用いることのできる，保護マトリックスに対する6つの基本操作を明確化した．これらの操作とはCREATE OBJECT, DELETE OBJECT, CREATE DOMAIN, DELETE DOMAIN, INSERT RIGHT, REMOVE RIGHTである．最後の2つの基本操作は特定のマトリックス・エレメントに権利を挿入したり，削除したりするものである．その一例としてドメイン1にFile6を読取る権利を与えることなどが挙げられる．

これら6つの基本操作は，保護コマンド(protection commands)として組み合わせることもできる．ユーザープログラムは保護コマンドを使用してマトリックスの変更が行える．この場合，基本操作を直接実行することはない．例えばシステムは新しいファイルの作成をコマンドを使って行い，その際ファイルが既存のものでなければ，新しい対象物を作成し，所有者にすべての権利を与えるという作業を行うこともある．また所有者が，システムに存在する各ユーザーに対して読取りの権利を与えるようなコマンドの存在も考えられる．このコマンドによって各ドメインの新しいファイルのエントリに"読取り"の権利を挿入することになる．

マトリックスは常に，認証ではなく，そのプロセスがどのドメインに属しているかによって権利を決定することになっている．マトリックスはシステムがどの様に規制を行うかを決定し，認証はどの様に管理するかを決定している．この2つの違いを明らかにしている例として，図5.27の簡単なシステムを考えてみよう．ここではドメインがユーザーに相当している(UNIXモデル

に類似している)．図5.27(a)では意図した保護方針が達成されている．すなわちHenryはmailbox7を読み書きでき，Robertはsecretの読み書きができ，3人のユーザーはすべてcompilerの読出しと実行が可能である．

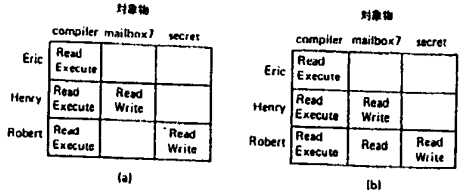

図5.27 保護マトリックス
(a)認証されている状態 (b)認証されていない状態

それではRobertが非常に頭がよく，マトリックスを図5.27(b)の様に変更するコマンドを発見したとしよう．ここでアクセス権のないmailbox7に対するアクセスを行った．読出しを行おうとするとオペレーティング・システムは図5.27(b)の状態が認められていないことを知らないため，その要求を実行してしまう．

使用可能なマトリックスの集合を2つの集合に区分できることがわかった．すなわち認証されている状態と，認証されていない状態の2つの集合である．多くの理論的な研究がもたらした疑問は，"最初の認証状態から，一連のコマンドを与えられても，システムが認証されていない状態にならないことを証明できるか" というものである．

すなわち与えられた機構(保護コマンド)が特定の保護方針の強化に適しているかという疑問である．方針の簡単な例として，軍事に用いられるセキュリティ手法を考えてみよう．各対象物は，秘密ではない，親展，秘密，または極秘扱いのいずれかである．各ドメイン(したがって各プロセスでもある)も，これら4つの機密保護レベルのいずれかに属する．セキュリティの方針では，以下の2つの決まりがある．

- プロセスが自分より高いレベルの対象物を読取ることはできないが，同等あるいはそれ以下のレベルの対象物は自由に読取ることができる．秘密レベルのプロセスは親展レベルの対象物を読取ることはできるが，極秘レベルの対象物を読み取る権利はない．

- プロセスは自分より低いレベルの対象物に情報を書込むことはできない．秘密レベルのプロセスは極秘レベルのファイルに書込むことはできるが，親展レベルのファイルには書き込めない．

軍事用語に置き換えて，兵士が親展，下仕官が秘密，将校が極秘レベルで活動を行っていると仮定すると，下仕官は兵士の書類を見ることはできても，将校の書類を勝手に見ることはできない．同様に下仕官は自分の知っていることを将校に報告することはできても，兵士はその権利を持たないため，彼らに報告してはならない．

この様な方針から，あるマトリックスの初期状態(各対象物のレベル決定方法も含む)から，一連のコマンドによりマトリックスが修正されていく過程で，システムの安全性を規定するにはどうしたら良いのであろうか．これを完全に行うことは非常に困難であることがわかった．多くの汎用システムは理論的に言って安全性に欠けている．この点に関する詳細はLandwehr(1981)とDenning(1982)の報告書を参照されたい．

5.5.5 隠れたチャネル

以上の項では，保護システムに対する正式なモデルの作成方法を見てきた．しかしここではその様なモデルを作成することがどれだけ無益であるかを学ぶことになる．特に安全性が絶対的に保証されているシステムにおいてさえ，理論的には通信が不可能なプロセス間の情報盗視は比較的簡単に行われている．これに関しては，Lampson(1973)の報告を参照されたい．

Lampsonのモデルには3つのプロセスが用いられており，主として大型のタイムシェアリング・システムに適用されている．最初のプロセスはクライアントで，第2プロセスであるサーバーになんらかの操作を要求するものである．クライアントとサーバーは互いに完全に信用はしていない．例えば，サーバーの働きがクライアントの租税申告書作成を手伝うことであるとする．クライアントは，サーバーが秘密裏に，例えば個人の収入リストを保存して，リストを誰かに売り付けたりしないかを配慮している．サーバーはクライアントがこの貴重な租税プログラムを盗まないかを配慮している．

第3のプロセスは敵側のスパイで，サーバーにクライアントのデータを実際に盗むよう入れ知恵する．スパイとサーバーは通常同じ所有者のものである．図5.28では3つのプロセスが示されている．この作業の目的はサーバーが合法的な手段でクライアントから入手した情報をスパイに漏らすことができないようなシステムを構築することである．Lampsonはこれを拘束問題(confinement problem)と呼んだ．

システム設計者の見方からいえば，サーバーがスパイに情報を漏らせないように，サーバーのカプセル化，または拘束を行えばよい．保護マトリックス手法を用いれば，スパイが読み出せるファイルにサーバーが書込みを行うことで，サーバーとスパイが交信することを防止することができる．またシステムのプロセス間通信機構を用いる場合も，同様なことを保証できるだろう．

しかし残念ながら，より狡猾な通信経路が存在している．例えばサーバーは以下の様にして2進のビット列を送信することができる．1ビットを送るためには，特定の時間間隔の間できる限りの演算処理を行う．0ビットを送る場合には，同じ時間の間，スリープ状態となる．

スパイはその応答時間を監視することによって，ビット列を検出する．一般的には，サーバー

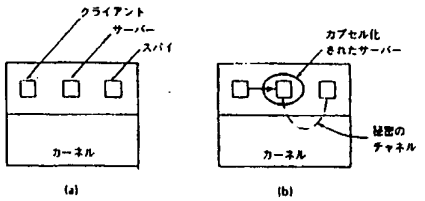

図5.28 隠れたチャネルによる情報漏洩
(a)クライアント，サーバーおよびスパイ・プロセス
(b)カプセル化されたサーバーでも隠れたチャネルを使ってスパイに情報を漏らすことができる

が1を送信している時よりも，0を送信している時の方が応答はよくなる．この通信経路は隠れたチャネル(covert channel)として知られている．図5.28(b)はこれを図示したものである．

いうまでもなく，隠れたチャネルは多くの雑多な情報を含むため雑音の多いチャネルではあるが，エラー訂正コード(例えばハミングコードあるいはより高度なもの)を使えば，雑音の多いチャネルでも確実に情報を送ることができる．エラー訂正コードを使用すると，すでに低い帯域幅をさらに低めることになるが，それでも重要な情報を漏洩することは可能である．対象物とドメインのマトリックスに基づいた保護モデルを使っても，この種の漏洩を防ぐことはできない．

CPUの利用率を使って情報を変調することだけが隠れたチャネルではない．ページング率も変調に利用することができる(ページフォルトが存在すれば1，存在しなければ0)．事実，何らかの方法でシステム性能を低下させるものであれば，変調に利用できる可能性がある．システムがファイルのロック機能を提供している場合，サーバーはファイルをロックして1と表し，ロックを解除したら0と表すことができる．通常，アクセスすることのできないファイルに関してもロック状態を知ることができる．

占有リソース(テープドライブ，プロッタなど)の入手と解除も同様に利用することができる．サーバーは資源を獲得して1を送信し，それを解除して0を送信する．UNIXではサーバーがファイルを作成して1を表示させ，それを削除して0を表示する．スパイはファイルの存在を確かめるためにシステムコールACCESSを用いることもできる．このコールはスパイがファイルを使用する権利を持っていなくても通用する．残念ながらこれ以外にも多くの隠れたチャネルが存在する．

Lampsonは，さらにサーバープロセスの所有者に情報を漏らす方法についても触れている．サーバープロセスは，クライアントに料金を請求するために，所有者にクライアントのためにどれだけの作業を行ったかを報告する．実際の演算料金が例えば100ドルで，クライアントの収入が53,000ドルであれば，サーバーは請求書に100.53ドルと書き込み，所有者に報告できる．

すべての隠れたチャネルを捜し出し，防御を行うことだけでも非常に困難である．実際にはそ

れをどうすることもできない．ランダムにページフォルトを発生するプロセスを採り入れたり，隠れたチャネルの帯域幅を低下させるためにシステム性能を下げることはあまり望ましくない．

5.6 MINIXファイルシステムの概要

他のファイルシステムと同様に，MINIXのファイルシステムもこれまで学んできた問題すべてに直面しなければならない．ファイルのための空間を割り当てたり，解除したり，ディスクブロックや空き空間の記録を取ったり，不正な使用からファイルを保護したり，といった問題が山積みされている．以降ではMINIXがこの問題をどの様な方法で解決しているかを詳しく解説していく．

本章では，前節までは一般性を重んじ，MINIXではなくUNIXを何度となく説明に用いた．もちろんこの2つの外部インターフェイスは全く同じである．ここではMINIXの内部構造について見ていくことにしよう．UNIXの内部構造に関しては，Thompson(1978)とBach(1986)の記述を参照されたい．MINIXのファイルシステムは単なるユーザー空間で動作するCのプログラムにすぎない(4章の図4.20参照)．ファイルを読み書きする際は，ユーザープロセスがメッセージをファイルシステムに送信し，必要な作業を要求する．ファイルシステムは指定された作業を行い，返答する．ファイルシステムは実質上，呼出し側と同じマシンの上で動いているネットワーク・ファイルサーバーにすぎない．

この様な構造は重要な意味を持っている．その1つは，ファイルシステムをMINIXの他の部分とは独立して修正，実験，検査できるようになることである．また，Cコンパイラを持つコンピュータであれば，ファイルシステム全体を移植し，そのコンピュータ上でコンパイルし，UNIXに類似した独立型のリモート・ファイルサーバーとして使用することもできる．この時変更しなければならないのは，システムによって異なるメッセージの送受信方法に関する記述だけである．

以下の項では，ファイルシステムの設計においてキーポイントとなる項目に関し，その概要を述べる．なかでもメッセージ，ファイルシステムの構造，ビットマップ，iノード，ブロックキャッシュ，ディレクトリ，パス名，プロセステーブル，特殊ファイル(およびパイプ)に関して詳しく見ていくことにする．これらに関する説明の後，ユーザープロセスがシステムコールREADを実行した時のそれぞれの状態を追跡し，各部分がどの様に連携しているのかを調べる．

5.6.1 メッセージ

ファイルシステムは，作業の要求を行う29種類のメッセージを受け付ける．そのうち2つを除いた残りすべてが，MINIXシステムコールに対するものである．2つの例外は，MINIX内のファイルシステム以外の部分で生成されるメッセージである．すべてのメッセージと，その引数，および結果を図5.29に示している．ファイルシステムは，メモリマネージャからも，メモリマネー

ジャが実行したいくつかのシステムコール(FORK や EXIT など)の結果をメッセージとして受取る．これらは主としてメモリマネージャによって扱われるため，図には示していない．

ファイルシステムの構造は，基本的にはメモリマネージャや入出力におけるそれと同じである．メッセージの到着を待ち待機するメインループを持つ．メッセージが到着すると，その型が取り出され，すべての型に対する手続きへのポインタを持つテーブルのインデックスとして使用し，該当する手続きが呼び出され，作業を実行し，ステータスを返す．ファイルシステムは次に呼出し側に返答を行い，次のメッセージを待つためにループの先頭に戻る．

メッセージタイプ	入力パラメータ	返答値
ACCESS	ファイル名，アクセスモード	状態(status)
CHDIR	新しいワーキングディレクトリの名前	状態(status)
CHMOD	ファイル名，新しいモード	状態(status)
CHOWN	ファイル名，新しい所有者，新しいグループ	状態(status)
CHROOT	新しいルートディレクトリの名前	状態(status)
CLOSE	クローズするファイルのファイル記述子	状態(status)
CREAT	作成するファイルの名前，モード	ファイル記述子
DUP	ファイル記述子(DUP2では2つ必要)	新しいファイル記述子
FSTAT	状態を調べるファイルのファイル名，バッファ	状態(status)
IOCTL	ファイル記述子，ファンクションコード，引数	状態(status)
LINK	連結させるファイル名，リンク名	状態(status)
LSEEK	ファイル記述子，変位，位置	新しいヘッド位置
MKNOD	特殊ファイル，ディレクトリの名前，モード，アドレス	状態(status)
MOUNT	特殊ファイル，マウント先，読取り専用フラグ	状態(status)
OPEN	オープンするファイル名，R/Wフラグ	ファイル記述子
PIPE	なし	ファイル記述子
READ	ファイル記述子，バッファ，バイト数	読み込んだバイト数
STAT	ファイル名，ステータスバッファ	状態(status)
STIME	現在時刻へのポインタ	状態(status)
SYNC	なし	常に真
TIME	現在時刻を格納する場所へのポインタ	実際の時間
TIMES	プロセスと子プロセスの消費CPU時間を格納するバッファへのポインタ	状態(status)
UMASK	モードマスク用のビット例	常に真
UMOUNT	マウント解除する特殊ファイル名	状態(status)
UNLINK	リンク解除するファイル名	状態(status)
UTIME	ファイル名，ファイルのタイムスタンプ	常に真
WRITE	ファイル記述子，バッファ，バイト数	書き込んだバイト数
REVIVE	再開させるプロセス	なし
UNPAUSE	停止させるプロセス	本文参照

図5.29 ファイルシステムが受け付ける主なメッセージ型
ファイル名の引数は，ファイル名を指すポインタである．返答値が状態(status)となっているものは，正常終了したか，または異常終了したかを示すものである．

5.6.2 ファイルシステムの構造

MINIX のファイルシステムは論理的，かつ自己完結な構成で，i ノード，ディレクトリ，およびデータブロックを持っている．ファイルシステムは，ブロック型デバイスであれば，フロッピーディスク，ハードディスク(またはその一部分)などのデバイスに作成することができる．どの様なデバイスに作成した場合でも，その構造はすべて同じである．図5.30は，127 の i ノードと，1K のブロックサイズを持つ，360K のフロッピーディスクを示している．これより大型のファイルシステムや，i ノード数，ブロックサイズの異なるファイルシステムにおいても同じ順序で同じ6つの構造を持っている．ただしその場合，それぞれの相対的な割合が異なるだけである．

各ファイルシステムはブートブロック(boot block)から始まる．MINIX 起動時，ハードウェアはブートブロックをメモリ内に読み取り，そこにジャンプする．すべてのディスクをブートデバイスとして使用することはできないが，構造に一貫性を持たせるため，すべての装置にブートブロックを設けている．システムがブートされると，ブートブロックは二度と使用されない．

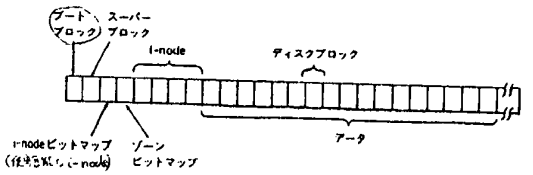

図5.30 フロッピーディスクの構造
127 の i ノードと 1K のブロックサイズから成る，典型的な360K のフロッピーディスクのディスク構造(2つの連続する512バイトセクタが1つのブロックとして扱われていることになる)．

スーパーブロック(super block)は，ファイルシステムの構造に関する情報を含んでいる．図5.31に，スーパーブロックを示している．

スーパーブロックの主な機能は，ファイルシステムに図5.30のそれぞれの大きさを伝えることである．ブロックサイズと i ノード番号を得れば，i ノードのビットマップと i ノードのブロック数を簡単に求めることができる．例えば1K のブロックであれば，ビットマップの各ブロックが1K バイト(8K ビット)を持っているため，最高 8,191 個の i ノードの状態を記録することができる(i ノード0の内容は常にすべて0となっており，意図的に使用されていない)．i ノードが10,000個存在する場合，2つのビットマップ・ブロックが必要である．i ノードは32バイトであるため，1K のブロックは最大32の i ノードを持つことができる．使用可能な i ノードが127個存在する時，それを保存するために4つのディスクブロックが必要となる．

ゾーンとブロックに関する詳細は後述することにし，ここではディスク容量が1，2，4，8単位

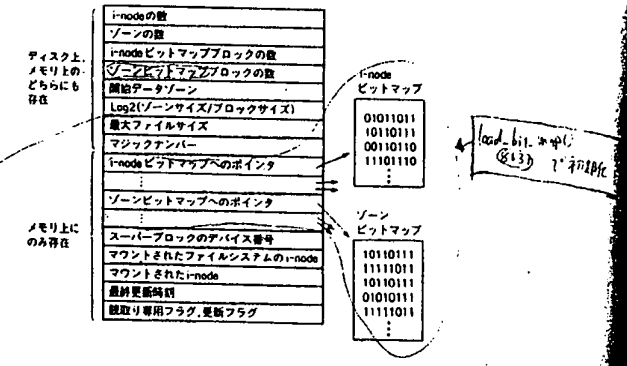

図5.31 MINIXのスーパーブロック

すなわち一般的には2^nブロック(ゾーン)単位で割り当てられるとだけ説明しておこう．ゾーン・ビットマップは空き容量をブロックではなく，ゾーン単位で記録している．標準の360Kフロッピーディスク版のMINIXでは，ゾーンとブロックサイズは同じ1Kであるため，最初の概算ではこれらの装置においてはゾーンはブロックと同じである．本章の終わりにおいて容量割当ての詳細にふれるまで，"ゾーン"を"ブロック"と考えてよい．

ゾーンあたりのブロック数は，スーパーブロックには保存されない．その必要が全くないからである．ブロック対ゾーンの比率の2を底とする対数を求め，ゾーンをブロックに，またはブロックをゾーンに変換する際のシフトカウントとして用いるだけでよい．例えばゾーンあたり8ブロックの場合，$\log_2 8=3$であるから，ブロック128を含むゾーンを検出するためには，128を右に3ビット分シフトし，ゾーン16を得なければならない．ゾーン0はブートブロックであるが，ゾーン・ビットマップはデータゾーンのみを含んでいる．

スーパーブロック内の情報は，色々な形式で使用されるため，そのどれにも対応できるよう，冗長的に提供されている．スーパーブロックに1Kを費やす場合，この情報を必要なすべての形式に対応できるようにしておく方が，実行中に頻繁に計算し直すよりも有意義である．例えばディスク上の最初のデータゾーンのゾーン番号は，ブロックサイズ，ゾーンサイズ，iノードの数，ゾーンの数から求めることができるが，スーパーブロック内に記録しておく方が速く求められる．スーパーブロックの残りの部分は使用されないため，別なワードをそこに配置しても何も影響はない．

MINIXをブートすると，ルート装置のスーパーブロックがメモリ内のテーブルに読み取られる．同様に他のファイルシステムがマウントされると，そのスーパーブロックもメモリ内に読み取られる．スーパーブロック・テーブルは，ディスク上には存在しない，いくつかのフィールド(装置情報など)と，読取り専用のマウントであるか否かを示すフィールド，そしてメモリ上の情報が

変更されたことを設定するフィールドが格納されている．

ディスクをMINIXファイルシステムとして利用するためには，図5.30の様な構造にしなくてはならない．提供しているユーティリティ・プログラムmkfsは，ファイルシステムを構築するためのものである．このプログラムは，以下の様なコマンド入力により実行することで，ドライブ1のフロッピーディスクに全ブロック360Kの新しいファイルシステムを構築することができる．

```
mkfs  /dev/fd1  360
```

このコマンドに対しては，新しいファイルシステムに格納するファイルとディレクトリをリスト表示したプロトタイプ・ファイルを指定することもできる．MS-DOSのディスケットなど，MINIXフォーマット以外でのファイルシステムのマウントを試みようとしても，MOUNTシステムコールはそのスーパーブロックのマジックナンバーなどを調べ，これを拒否する．

5.6.3 ビットマップ

MINIXでは，2つのビットマップを用いてどのiノードが空き状態となっているかを調べている(図5.31参照)．システムをブートすると，ルートデバイスのスーパーブロックとビットマップがメモリにロードされる．前述したように，メモリ内のスーパーブロック・テーブルは，ディスク上に存在しない，いくつかのフィールドを持っている．このフィールドの1つとしてiノードのビットマップ・ブロックへのポインタの配列が設定されている．ビットマップが4ブロックから成る時，配列は，4つのポインタから構成される．

ファイルを削除する時に，どのブロックのビットマップに解放されるiノードが含まれているかを求めることは，ポインタの配列により簡単に行える．ブロックを検出すると，解放するiノードに相当するビットが1に設定される．ゾーン・ビットマップでも同様のポインタ配列が使用される．

ファイルを作成する場合，ファイルシステムはビットマップ・ブロックを1つずつ走査し，空きiノードを検出しようとする．そしてこのiノードが新しいファイルに割り当てられることになる．ディスクの各iノードスロットが満杯になると，検出ルーチンは0を返す．このため，iノード0を使用しなかったのである(mkfsによって新しいファイルシステムを作成すると，iノード0を0にして，ビットマップの最下位ビットを1に設定する．これによってファイルシステムはiノード0の割当てが行えなくなる)．

この様な背景を基に，ゾーンとブロックの違いについて触れたい．ゾーンの概念は，同じファイルに属するディスクブロックを同じシリンダ上に必ず存在させ，ファイルを連続的に読み取る場合の性能を向上させるというものである．そこで複数のブロックを同時に割り当てられるような方法が取られた．例えばブロックサイズが1Kで，ゾーンサイズが4Kの場合，ゾーン・ビットマップはブロックではなく，ゾーンに関する記録をとる．20Mのディスクは4Kのゾーンを5,000

個持つため，ゾーン・ビットマップには5Kビットが必要となる．

多くのファイルシステムでは，ブロックが用いられている．ディスク転送は必ず1ブロックごとに行われ，バッファ・キャッシュもブロック単位で提供されている．物理的なディスクアドレス(例えばゾーン・ビットマップおよびiノードなど)を記録する部分だけがゾーンを関知している．

ゾーンを使用するもう1つの理由は，ディスクアドレスを16ビットに制限，統一することによって，間接ブロックにできるだけ多くの情報を保存することができるからである．しかし16ビットのゾーン番号と，1Kのゾーンを用いた場合，65,000個のゾーンまでしかアドレス指定することができない．つまりディスクを65Mに制限していることになる．ディスクが大きくなっても，ブロックサイズはそのままで，ゾーンを2Kまたは4Kに切り換えることが簡単に行える．大半のファイルは1Kより小さいため，ブロックサイズを増やすことによって，多くの空きを持つブロックの読み書きをすることになり，バッファ・キャッシュ内にそれを保持するために貴重な主記憶を無駄使いし，さらにディスク容量を浪費することになる．もちろんゾーンサイズが増加すると，ディスク容量の浪費は多くなるが，大きなゾーンは大きなディスクにしか必要ないため，ディスク容量の効率に関する問題はそれほど深刻なものではない．

ゾーンはこれ以外にも予期せぬ問題を引き起こすこともある．再度4Kのゾーンと1Kのブロックを用いた例を考えてみよう．1Kの大きさを持つファイル，すなわち1ゾーンが割り当てられたファイルを仮定する．1Kと4Kの間のブロックにはごみ情報(前所有者の忘れ物)が含まれているが，iノードにはファイルサイズとして1Kと記録されているため，特に問題はない．ファイルの終端を超えて読出しを行うと，必ず0のカウントがデータなしで返されてくる．

ここで32768番地にシークし，1バイト書き込むとしよう．ファイルサイズはこれによって32769に変更される．データを読み出すために1Kのシークを連続的に行うことにより，そのブロックに以前含まれていた情報を読み取ることができる．これは明らかに深刻なセキュリティの規定違反である．

そこで解決策として，書込みがファイルの終端を超えて行われた時にこの様な状態の発生を確認し，以前は使用されていたゾーン内のブロックのうち，まだ割当てが行われていないものをすべて0にする．この様な状況が発生することは滅多にないが，少々システムが複雑になってもこの問題を避けて通るわけにはいかない．逆説的にいえば，この犠牲を払ってゾーンを利用するだけの価値があるかどうかは不明である．通常のMINIXでは，ゾーンサイズとブロックサイズをどちらも1Kに設定しているため，問題はない．

5.6.4 iノード

MINIXにおけるiノードの構造を図5.32に示している．いくつかの点でUNIXのiノードとは異なっている．まずディスクブロックへのポインタが短い(MINIXは2バイト，UNIXは3バイト)．2番目に保存しているポインタ数が少ない(MINIXは9，UNIXは13)．3番目にMINIXでは時刻を1つしか記録していないが，UNIXでは3つ記録している．さらにMINIXでは，リ

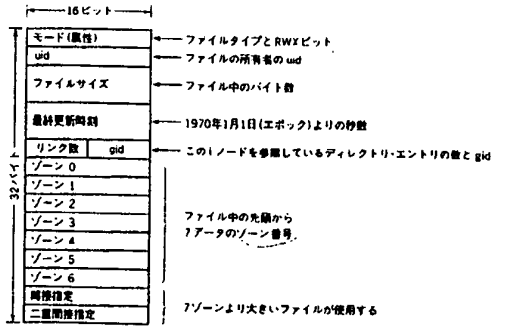

図5.32 MINIXのiノード

ンク数とgidフィールドが1バイトに削減されている．この様な変更を行った結果，サイズが64から32バイトになり，iノードの保存に必要なディスクとメモリ容量が削減できる．

ファイルをオープンすると，そのiノードが検索され，メモリ内のiノードテーブル内に置かれ，ファイルをクローズするまでメモリ内に保存される．iノードテーブルには，デバイスの番号を示すiノードのフィールドなど，ディスク上には存在しないフィールドがいくつか含まれている．ファイルシステムはこれらのフィールドを使い，メモリ上で情報が修正され場合には，どこにそれを書き込めばよいかを判断する．またiノードごとのカウンタも記録されている．同じファイルが複数回オープンされても，メモリ内にはiノードのコピーが1つだけ保存され，カウンタはファイルがオープンされるたびに増加され，クローズされるたびに減少される．カウンタが0になった時に限り，iノードがテーブルから削除されることになる(その時点で修正されていれば，ディスクに書き込まれる)．

ファイルに割り当てられているiノードの主な機能は，データブロックの位置を知らせることである．最初の7つのゾーン番号は，iノード上に直接用意されている．標準のMINIXでは，ゾーンとブロックはどちらも1Kなので，7Kまでのファイルでは間接ブロックが必要ない．7K以上になった場合，間接ブロックが必要になるため，図5.7の手法を用いる．ただしMINIXでは，単一間接と二重間接ブロックのみを使用している．1Kブロックとゾーン，そして16ビットのゾーン番号を用いた場合，単一間接ブロックは512個のエントリを保存することができる．これは0.5Mバイトの容量である．二重間接ブロックは，512個の単一間接ブロックを指し，最大256Mバイトまでとなる(実際には16ビットのゾーン番号と，1Kゾーンを用いた場合，64Kのゾーン，すなわち64Mバイトしかアドレス指定できないため，この制限値を超えることはない．これより大きなディスクの場合は，ゾーンを2Kに変更しなければならない)．

iノードは，さらにモード情報も含んでおり，ファイルの種類(通常ファイル，ディレクトリ，ブロック型特殊，キャラクタ型特殊，あるいはパイプ)，保護状態とSETUID，SETGIDビットを提供する．iノードレコード内のフィールドであるリンク数は，iノード上を指しているディレクトリ・エントリの数を表している．ファイルシステムは，このリンク数により，ファイルに割り当てられている領域を解放するかどうかを判断している．このフィールドを，ファイルの現在のオープン回数を知らせるカウンタ(ディスク上ではなく，メモリ内のiノードテーブルにのみ存在)と混同してはならない．

5.6.5 ブロックキャッシュ

MINIXでは，システム性能を向上させるために，ブロックキャッシュを用いている．キャッシュはバッファの配列として実現される．各バッファはポインタを持つヘッダ，カウンタ，フラグ，そして1つのディスクブロック分の空間を持つ本体部分から構成されている．全ブロックは図5.33の様に最も最近使用したもの(MRU)から，最も昔に使用したもの(LRU)の順に，二重に連結されたリストを使ってつながれている．

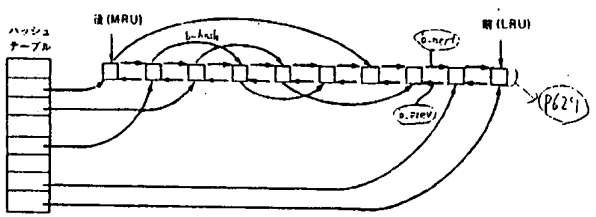

図5.33 ブロックキャッシュの連結されたリスト

さらに特定のブロックがキャッシュ上にあるか否かを即座に判定する方法として，ハッシュテーブルを使用することもできる．ハッシュコードkを持つすべてのブロックは，ハッシュテーブル内のエントリkが指す単一に連結されたリスト上につながれる．現在のところ，ハッシュ機能は単にブロック番号から下位nビットを抽出しており，このため異なるデバイスのブロックが同じハッシュチェーン内に存在している．

ファイルシステムがブロックを必要とする時は，手続きget_blockを呼び出し，そのブロックのハッシュコードを算出して，ハッシュリストを検索する．ブロックが検出された場合，ブロックのヘッダ内のカウンタが減少され，ブロックが使用中であることを示す．そのあとそれを指すポインタが返される．ブロックがチェイン検出できなかった場合，LRUリストを検索し，キャッシュから除去するブロックを決定する．リストの前方のブロック(最も昔に使用されたブロック)がカウント0となっている場合，そのブロックが選択される．そうでなければ次のブロックが調べられ，といった具合である．ビットマップなど，ブロックによっては使用頻度に関わらず使用中に除去してはならないものもあるため，カウンタの検査は重要である．

除去するブロックを選択した後，そのヘッダ内の別のフラグが調べられ，ブロックを読み取った後で修正が行われたかどうかを確認する．修正が行われていればディスクに書き込まなくてはならない．そして必要なブロックを，ディスクタスクにメッセージを送信することによって読み取る．ファイルシステムはブロックの読取りが終了するまで保留状態となり，終了した時点で実行を継続し，ブロックを指すポインタが呼出し側に返される．

ブロックを要求した手続きが処理を終了すると，別な手続きput_blockを呼び出し，ブロックを解放する．put_blockに対する引数のうちいずれかが，解放されるブロックのクラス(例えばiノード，ディレクトリ，データなど)を示している．クラスによって以下の2つの重要な決定がなされる．

- ブロックをLRUリストの前に置くか，後ろに置くか
- ブロックを即座に書き込むか否か(ブロックに修正が行われている場合)

二重間接ブロックなど，しばらく必要のないブロックはリストの前方に置かれ，次回空きバッファが必要になった時に要求できるようにしておく．すぐに必要になるブロックは，LRU形式に従いリストの最後に置かれる．

ディレクトリが修正された場合，クラッシュ発生時のファイルシステム破損の危険性を減らすため，即座にディスクへ書き込まれる．通常の修正済みデータブロックは，(1)LRUチェーンの先頭に来て除去されるか，(2)システムコールSYNCが実行された時，のうち，いずれかのイベントが発生するまで書き換えられない．

ブロックが修正されたことを示すヘッダ上のフラグは，ブロックの要求と使用を行ったファイルシステム内部の手続きによって設定される．手続きget_blockとput_blockは，連結されたリストの操作のみを行う．ファイルシステムがどのブロックをどの様な理由で必要としているかには，全く関知していない．

5.6.6 ディレクトリとパス

ファイルシステム内のもう1つ重要なサブシステムとして，ディレクトリとパス名の管理がある．OPENなど大半のシステムコールは，ファイル名を引数として持っている．必要であるのは，そのファイルのiノードであるため，ファイルシステムはディレクトリ・ツリーからそのファイルを検出し，iノードの位置を調べなくてはならない．

MINIXのディレクトリは1つのファイルに対して16バイトのエントリから構成されている．最初の2バイトは16ビットのiノード番号に，残りの14バイトはファイル名に使用されている．パス/user/ast/mboxを検索する場合，システムはまずルートディレクトリ内からuserを検索

し，次に/user 内の ast を検索し，最後に/user/ast 内の mbox を検索する．実際の検索はパスの要素それぞれに対し，1回に1つずつ行われる．図5.11はこれを示したものである．

これをさらに詳しく説明すると，標準的な MINIX の構成では/usr をフロッピーディスク0(システムファイル)に，/user をフロッピーディスク1(ユーザーファイル)に用いている．以下の例では典型的なユーザーディレクトリとして/user/ast を使用している．

唯一複雑であるのは，マウントされたファイルシステムに遭遇した場合である．その仕組みを知るために，マウント方法を見てみなければならない．ユーザーが以下の入力を端末から行ったとしよう．

```
/etc/mount  /dev/fd1  /user
```

フロッピーディスク1に格納されているファイルシステムは，ルート・ファイルシステムの/user にマウントされる．マウント前とマウント後のファイルシステムを図5.34に示す．

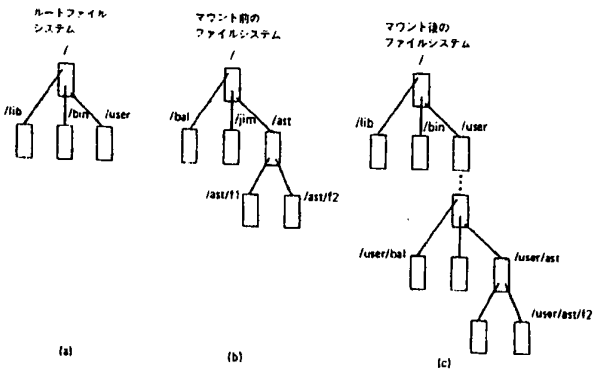

図5.34 ファイルシステムのマウント
(a)ルート・ファイルシステム (b)マウント前のファイルシステム
(c)(b)のファイルシステムを/user にマウントした結果

マウント作業全体の鍵を握るのが，マウント成功後に/user のiノードに設定されるフラグである．このフラグはiノードにマウントが行われたことを示すものである．MOUNT システムコールは，新しくマウントされたファイルシステムのスーパーブロックを，super_block テーブルにロードし，そこに2つのポインタを設定する．さらに新しいファイルシステムのルートのiノードをiノードテーブルに置く．

図5.31を見ると，メモリ内のスーパーブロックが，マウントされたファイルシステムに関係のある2つのフィールドを含んでいることがわかる．「マウントされたファイルシステムのiノード」は，新しくマウントされたファイルシステムのルートiノードを指すように設定される．次のフィールドである「マウントされたiノード」は，マウントされたiノード，つまりここでは/user のiノードを指すように設定される．これら2つのポインタはマウントされたファイルシステムをルートに接続するために，その2つの間で「接着剤」の働きをするものである(図5.34(c)では点線で示されている)．

/user/ast/f2 というパス名を検索する場合，ファイルシステムは/user 内のiノードのフラグを調べ，/user 上にマウントされたファイルシステムのルートiノードを引き続き検索しなければならないことがわかる．それではこのルートiノードはどの様にして検出するのだろうか．

答えは簡単である．システムはメモリ内のスーパーブロックを，「マウントされたiノード」フィールドが，/user を指すものが検出できるまで検索する．これが/user にマウントされたファイルシステムのスーパーブロックである．スーパーブロックを検出すると，他のポインタに従ってマウントされたファイルシステムのルートiノードを簡単に検出することができる．ここでファイルシステムは検索を続けることができる．例えばフロッピーディスク1のルートディレクトリ内の ast を検索できる．

5.6.7 ファイル記述子

ファイルがいったんオープンされると，ファイル記述子がユーザープロセスに返され，以降の READ および WRITE コールで利用できるようになる．この項では，ファイルシステム内のファイル記述子の管理方法について考察していく．

カーネルやメモリマネージャ同様に，ファイルシステムはプロセステーブルの一部分をアドレス空間内に保存する．このうち，3つのフィールドが特に重要である．最初の2つはルートディレクトリとワーキングディレクトリのiノードを指すポインタである．図5.11に示したパス名の検索は，パス名が絶対的なものであるか，相対的なものであるかにより，ルートディレクトリかまたはワーキングディレクトリのどちらかから検索を開始する．これらのポインタは，CHROOT および CHDIR システムコールを用いて，新しいルート，またはワーキングディレクトリを指すよう変更することができる．

プロセステーブル内で次に重要なフィールドは，ファイル記述子の番号をインデックスとした配列である．これはファイル記述子が与えられた時に，そのファイルの場所を示すためものである．単純に考えると，ファイル記述子 k に対するファイルを示すのみであれば，この配列の k 番目にそのファイルのiノードを示す情報を持つだけで十分のように思われる．iノードはファイルがオープンされるとメモリに取り出され，クローズされるまでそこに保存されるため，この方法が利用できそうである．

残念ながらこの様な単純な方法は，MINIX では(UNIX においても)，ファイルの共有が行われる場合もあるためうまくいかない．次に読み書きを行うバイトを示すための32ビット長の値が，

それぞれのファイルに対してに用意されるため，問題が生じる．この値はファイル位置(file position)またはファイルポインタと呼ばれ，LSEEKシステムコールを用いて変更することができる．つまり問題となるのはファイル位置の保存場所である．

解決方法として考えられるのは，iノードにファイル位置を保存することである．残念ながら複数のプロセスが同時に同じファイルをオープンすると，それぞれのファイル位置を持つことになる．あるプロセスがLSEEKを実行し，次に別のプロセスが読出しを行うような場合に問題が発生してしまう．したがって，iノードにはファイル位置を保存できないのである．

それではプロセステーブルに置けばどうだろう．ファイル記述子の配列と平行にもう1つ配列を設け，各ファイルの現在の位置を格納したらどうだろう．この方法も役に立たないのだが，その理由はもっと明白である．基本的には，FORKシステムコールの意味論が問題の原因となる．プロセスがフォークすると，親子のプロセスが各オープン・ファイルの現在位置を示す1つのポインタを共有する必要がある．

問題をわかりやすくするために，出力がファイルにリダイレクトされたシェルスクリプトの例を考えてみよう．シェルが最初のプログラムをフォークすると，標準出力に対するそのファイル位置は0となる．この位置は子プロセスにも受け継がれ，例えば1Kの出力を書き込んだとする．子プロセスが終了すると，共有するファイル位置は1Kになる．

ここでシェルはさらにシェルスクリプトを読み取り，別な子プロセスをフォークする．2番目の子プロセスはシェルからファイル位置として1Kを受け継がなくてはならないため，最初のプログラムが終了した場所から書込みを開始する．シェルがその子プロセスとファイル位置を共有しなければ，2番目のプログラムは出力内容を最初のプログラムに追加するのではなく，上書きしてしまうことになる．

結論として，ファイル位置をプロセステーブルに置くことは不可能であることがわかった．MINIXでは解決策として新しい共有テーブルfilpを設け，すべてのファイル位置を保存することにした．その用法を図5.35に示している．ファイル位置を実際に共有させることにより，FORKは正しく実現され，シェルスクリプトも正しく動くことになる．

filpテーブルに格納しなければならないものは，共有のファイル位置であるが，iノードへのポインタも格納しておくと便利である．ここにiノードへのポインタが格納されていると，プロセステーブル内のファイル記述子配列内には，filpエントリを指すポインタだけを保存するのみでよい．filpエントリにはさらにそれを使用しているプロセスの数も含まれており，ファイルシステムがそのエントリを使用する最後のプロセスが終了した場合，そのスロットを再利用できるようにしている．

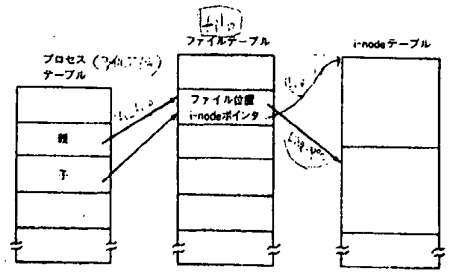

図5.35 親子間のファイル共有

5.6.8 パイプと特殊ファイル

パイプと特殊ファイルは，基本的な点で通常のファイルと異なっている．プロセスがディスクファイルから読み書きを行う時，操作は必ず2,300ミリ秒間のうちに終了する．最悪の場合いくつかのディスクアクセスを必要とするかもしれない．パイプから読み出す場合はこれとは異なっている．パイプが空であれば，読出しを行おうとする者は，誰かがパイプ内にデータを送り込むまで待機しなければならず，場合によってはそれまで何時間も要することもある．同様に，端末から読み取る場合も，プロセスは誰かが何かを入力するまで待たなければならない．

結論として，ある要求が終了するまで次の処理は行わないというファイルシステムの通常の規定は効力を持たないことになる．これらの要求をいったん中断し，後で再開することが必要である．プロセスがパイプから読み書きをする場合，ファイルシステムはパイプの状態を即座に調べ，操作が完了できるかを確認する．完了可能な操作であれば処理を行い，完了不可能であれば，ファイルシステムはシステムコールの引数をプロセステーブル内に保存し，時期を見てプロセスを再開できるようにする．

ファイルシステムは，特別な操作を行わなずに呼出し側をいったん停止させることができる．返答を行わなければ，呼出し側は返答待機状態のままブロックしてしまうからである．したがってプロセスをいったん停止した後で，ファイルシステムはメインループに戻り，次のシステムコールを待つ．別なプロセスがパイプの状態を変更し，保留中のプロセスを完了できるようになった時，ファイルシステムはフラグを設定し，次のメインループで保留中のプロセスの引数をプロセステーブルから取り出し，コールを実行する．

端末や他のキャラクタ特殊ファイルでは，若干状況が異なる．特殊ファイルのiノードには，2つの番号が格納されている．主デバイス番号と，副デバイス番号である．主デバイス番号は，デバイスクラスを示すものである(例えばRAMディスク，フロッピーディスク，ハードディスク，端末など)．それをファイルシステムテーブルに対するインデックスとして使用し，該当するタスク(すなわち入出力ドライバ)番号に対応付ける．この様に，主デバイス番号によって呼び出す入出

力ドライバを決定するのである．副デバイス番号は，ドライバに対し引数として渡される．これは使用するデバイスとして，例えば端末2やドライブ1などを指定するものである．

プロセスが特殊ファイルからの読取りを行う時，ファイルシステムはファイルのiノードから主および副デバイス番号を取り出し，主デバイス番号を該当するタスク番号への対応付けのために，ファイルシステムテーブルのインデックスとして利用する．タスク番号を得ると，ファイルシステムはタスクにメッセージを送る．メッセージには引数として副デバイス番号，実行する操作，呼出し側のプロセス識別子，バッファ・アドレス，そして転送するバイト数が含まれている．形式は，POSITIONフィールドが使用されていないという点をのぞき，3章の図3.15と同じである．

ドライバが即座に作業を実行できる場合(例えば入力行がすでに端末から入力されている場合など)，データをドライバの内部バッファからユーザー用バッファにコピーし，作業の終了を伝える返答メッセージを送る．ドライバはデータをファイルシステムにはコピーしない点に注意されたい．ブロック型デバイスからのデータはブロックキャッシュを通過するが，キャラクタ型特殊ファイルはそこを通過しない．

ドライバが作業を実行できない場合，メッセージ・パラメータをその内部テーブルに記録し，即座にシステムコールを完了できないことを伝える返答メッセージをファイルシステムに送る．この時点で，ファイルシステムは誰かが空のパイプから読出しを行おうとしていることを発見した場合と同じ状況になる．プロセスが保留状態になったことを記録し，次のメッセージを待機する．

ドライバが処理を完了できるだけのデータを得た時，それをブロック中のユーザーのバッファに転送し，ファイルシステムにその旨を伝えるメッセージを送る．ファイルシステムはここでユーザーに対し，ブロック解除のための返答メッセージを送り，転送されたバイト数を報告すればよい．

5.6.9 システムコールREAD

後述するが，ファイルシステムの大半のコードはシステムコールを実行するために書かれている．したかって最も重要なシステムコールであるREADの仕組みを述べることにより，以上の概要説明を締めくくることにする．

ユーザープログラムが以下の様なステートメントを実行し，通常のファイルを読み取ろうとした．

```
n=read(fd, buffer, nbytes);
```

ライブラリ関数readは，3つの引数と共に呼び出されている．この結果，3つの引数を持つメッセージが，メッセージ型としてREADに対するコードと共に作成され，メッセージがファイルシステムに送信され，返答を待ってブロックする．メッセージが到着すると，ファイルシステムはメッセージ型をテーブルに対するインデックスとして用い，読出しを行う手続きを呼び出す．

この手続きはファイル記述子をメッセージから取り出し，それをfilpエントリと，読み出すファイルのiノード検出に用いる(図5.35参照)．そして要求は1ブロック内に収まるように分割される．例えば現在のファイル位置が600で，1Kバイトが要求されている場合，要求は600から1023と，1024から1623の2つに分割される(1Kブロックと仮定して)．

これらを1つずつ検査し，そのブロックがキャッシュ内に存在するか調べる．存在しなければ，ファイルシステムは最も昔に使用したバッファのうち現在使用されていないものを選択し，その内容が変更されていれば書き直すように指示したメッセージをディスクタスクに送る．そして次にディスクタスクに読み取るブロックを要求する．

ブロックがキャッシュに入ると，ファイルシステムはデータをユーザーバッファ内の該当する場所にコピーするよう指示したメッセージをシステムタスクに送る(すなわちバイト600から1023をバッファの先頭に，バイト1024から1623をバッファ内の424の位置に転送する)．コピーが終了したら，ファイルシステムは返答メッセージをユーザーに送り，コピーしたバイト数を伝える．

ユーザーがこのメッセージを受け取ると，ライブラリ関数readは返答コードを取り出し，呼出し側に関数値としてそれを返す．

5.7 MINIXファイルシステムの実現

MINIXファイルシステムは比較的大きいが(Cで100ページを超える)，かなりわかりやすいものとなっている．システムコール実行のための要求が行われ，実行され，返答が行われる．本節では1ファイルごとにそれを見ていくことにしよう．特に重要と思われるところは解説を行っている．コード自体にも多くの注釈が含まれているので参照されたい．

5.7.1 ヘッダファイル

カーネルとメモリマネージャの様に，ファイルシステムにも各種のデータ構造とテーブルを定義するヘッダファイルが存在する．それではそのファイルシステムから見ていくことにしよう．

ファイルconst.h(7500行目)は，テーブルサイズやフラグなど，ファイルシステム全体で使用される定数を定義するものである．なかにはNR_BUFSおよびNR_BUF_HASHの様に，システム性能をチューニングするために変更できるものもある．その他BOOT_BLOCKおよびSUPER_BLOCKなどは，性能とは関係がない．

次のファイルbuf.h(7550行目)では，ブロックキャッシュの定義を行っている．配列bufには，すべてのバッファが保存される．各バッファにはブロックb，そして多くのポインタ，フラグ，カウンタが含まれている．データ部分は5つの型のユニオンとして宣言されている(7565行目)．これは場合によってはブロックを文字列として，またはディレクトリなどとして参照するために効

果的だからである．

バッファ3のデータ部分を文字配列として参照する場合，buf[3].bは共用体全体を参照し，そこからフィールドb__dataを選択するためには，buf [3] .b.b__dataとするのが正しい方法である．この構文は正しいが，多少面倒であるため，7588行目ではマクロb_dataを定義し，buf[3].b_dataと略記する．b__data(ユニオンのフィールド)では2つの下線を使用しているが，b_data(マクロ)では識別しやすいように1つしか使っていない点に注意されたい．ブロックの別な部分をアクセスするためのマクロは，7588から7592行目に含まれている．

このファイルでもう1つ注目したいのは，すべての配列と変数に対してEXTERNを用いていることである．このファイルがコードファイルに含まれている時，EXTERNは0009行目で定義されているように，externに置き換えられる．しかしファイルtable.cでは，保存領域の割当てを行うためにヌルストリングとして定義されている．Cでは(Kernighan and Ritchie, 1978, 原書p.206)，大域変数は1つのファイルを除くすべてのファイルでexternとして宣言されなくてはならないことになっている．この点を理解していないコンパイラやプログラマが多いのは残念なことである．これと同じような問題がカーネルとメモリマネージャにも見られた．

ファイル最後のマクロ(7601から7610行目)は，ブロックの種類を定義するものである．ブロックが使用後にバッファ・キャッシュに戻された時，これらの値のいずれかが与えられ，キャッシュマネージャはそれによってそのブロックをLRUリストの先頭または後ろのどちらに置くか，またディスクにすぐ書き込むか否かを判断する．

ファイルdev.h(7650行目)は，dmapテーブルの定義を行うものである．テーブル自身は初期値と共にtable.cで宣言されており，複数のファイルに含まれないようになっている．dev.hが必要な理由はここにある．テーブルは主デバイス番号と該当するタスクの対応付けを行う．

ファイルfile.h(7700行目)は現在のファイル位置と，iノードのポインタを格納するための中間テーブルを持つ(図5.35)．さらにファイルが読取り，書込み，あるいはその両方，のいずれを行うためにオープンされたのか，そして現在そのエントリを指しているファイル記述子の数を格納する．

プロセステーブルのファイルシステム部分はfproc.h(7750行目)に記述されている．モードマスク，現在のルートディレクトリとワーキングディレクトリに対するiノードを指すポインタ，ファイル記述子の配列，uid, gid, および端末番号が含まれている．残りのフィールドは例えば空のパイプを読み出そうとして途中で保留になっていたシステムコールの引数を保存するために使われる．フィールドfp_suspendedおよびfp_revivedは，実際には1ビットしか必要としないが，大半のコンパイラはビットフィールドよりも文字に対して質の良いコードを作成するため，この様にしている．

次は大域変数を持つファイルglo.hである．受け取るものと返答メッセージに対するメッセージ・バッファも，システムスタックと共にここに存在する．MINIXのブート後，ファイルシステムが初めて起動されると，非常に小さなアセンブラ手続きがスタックポインタをfstackの最上部

に設定する．

次はinode.hのiノードテーブルである(7850行目)．何度も触れたように，ファイルをオープンすると，そのiノードがメモリ内に読み込まれ，ファイルをクローズするまでそこに保存される．これらのiノードはこのテーブルに保存される．以上で，大半のフィールドの意味は理解できたはずである．ただしi_seekに関して，若干の説明を加えておこう．最適化の一環としてファイルシステムは，ファイルが連続して読み出されることに気づくと，要求がくる前にブロックをキャッシュに読み込もうとする．ランダムアクセス・ファイルでは，先読みの機能は提供されていない．LSEEKコールが実行されると，フィールドi_seekが設定され，先読みを禁止する．

フィールドparam.h(7900行目)は，メモリマネージャ内の同じ名前のファイルと似ている．引数を含むメッセージ・フィールドの名前を定義し，例えばメッセージ・バッファmの1フィールドを選択するm.m1_p1の代わりに，bufferとして参照できるようにする．

super.h(7950行目)ではスーパーブロック・テーブルの宣言が行われている．システムがブートされると，ルートデバイスのスーパーブロックがここへロードされる．ファイルシステムのマウント時にも，そのスーパーブロックがここに配置される．

最後に型の定義がtype.h(8000行目)で行われている．2つの型，すなわちディレクトリ・エントリとディスクiノードが定義されている．

5.7.2 テーブル管理

主要な各テーブル，すなわちブロック，iノード，スーパーブロックなどのテーブルを管理する手続きは，それぞれが1つのファイルにまとめられている．これらの手続きはファイルシステムの他の部分でも頻繁に使用され，テーブルとファイルシステム間の主要なインターフェイスとなっている．したがって，そのファイルシステムコードから学習していくことにしよう．

■ ブロック管理

ブロックキャッシュはファイルcache.cの手続きによって管理されている．このファイルには図5.36で示した5つの手続きが含まれている．最初の手続きget_block(8079行目)はファイルシステムによるデータブロック取得方法ととして標準的なものである．ファイルシステム手続きがユーザーデータブロック，ディレクトリ・ブロック，スーパーブロック，その他各種のブロック

get_block	読み書きブロックを得る
put_block	直前のget_blockによって要求されたブロックを返す
alloc_zone	新しいゾーンを割り当てる（ファイル成長）
free_zone	ゾーンの解放（ファイルの削除）
rw_block	ディスク，キャッシュ間のブロック転送
invalidate	デバイス用の全キャッシュを一掃する

図5.36 ブロック管理用の手続き

を読み取る場合，デバイスとブロック番号を指定して get_block を呼び出す．

get_block を呼び出すと，最初にブロックキャッシュを調べて要求したブロックが存在しているかどうかを確かめる．もし存在していれば，それを指すポインタを返す．存在しない場合には，ブロックを読み出さなければならない．キャッシュ内のブロックは連結されたリスト NR_BUF_HASH(32)上につながれている．それぞれのリストにつながれているブロックはすべてブロック番号の最後が同じ5ビットの並び，すなわち 00000，00001…11111 を持っている．

8099行目のステートメントは，要求したブロックがキャッシュ内に存在するならば，そのブロックが属していることになっているリストの先頭を bp が指すよう設定する．8101行目のループはこのリストを検索し，ブロックを検出する．存在すれば8106行目で呼出し側に返される．

ブロックがリスト上に存在しない場合は，キャッシュ内にも存在しないことになるため，最も昔に使用されたキャッシュのうち，現在使用されていないものが用意される．ビットマップなど，現在使用されているブロックを除去してはならない．選択したバッファはハッシュチェーンから削除される．新しいブロック番号で使用するために別なハッシュチェーン上に移すためである．内容が修正されている場合は8139行目でディスクに書き込まれる．

バッファが使用可能になると，即座に新しい引数が設定され，ブロックがディスクから読み取られる．これには例外が1つある．ファイルシステムが1つのブロック全体を書き込もうとする場合，最初に古い内容を読み出すのは無駄である．この場合は，ディスクの読取りが省略される(8149行目)．新しいブロックが読み取られた後，get_block は呼出し側にそれを指すポインタを返す．

ファイルシステムがファイル名の検索のために，一時的にディレクトリ・ブロックを必要としていると仮定しよう．get_block を呼び出してディレクトリ・ブロックを得る．そのファイル名を検出した後，put_block を呼び出して(8157行目)ブロックをキャッシュに返却する．これにより，別なブロックにおいて必要になった時，このバッファを利用できることになる．

手続き put_block は，新らたに返却されたブロックを LRU リスト上に置き，場合によってはディスクに書込みを行う．まず最初に(8189行目)ブロックを LRU リストの現在位置から除去する．次にブロックの種類を表すために呼出し側が指定したフラグ block_type を基に，LRU リストの先頭，または最後にそれを置く．しばらく必要ないと思われるブロックは先頭に置かれ，そこですぐに再利用される．すぐに必要となるブロックは最後に置かれ，しばらくそこで待機する．

ブロックが LRU リスト上に再配置されると，ディスクにブロックの書込みをすぐに行う必要があるかが検査される(0057から8225行目)．i ノード，ディレクトリ・ブロック，その他ファイルシステムの機能自体に関係のあるブロックはこれに属するため，その時点でディスクに書き込まれる．

ファイルが成長すると，場合によっては新しいゾーンを割り当て，新しいデータの保存に備えなくてはならないことがある．手続き alloc_zone(8235行目)は，新しいゾーンの割当てを行うものである．ゾーン・ビットマップを検索し，空きゾーンを検出する．現在のファイルのゾーン0に

近いゾーンを検索し，ファイルのゾーンをひとまとめにしようとする．ビットマップ内のビット番号と，ゾーン番号の対応は8268行目で行われている．ビット1が最初のデータゾーンに相当することがわかる．

ファイルを削除すると，そのゾーンがビットマップに返却される．free_zone(8275行目)はこれらゾーンの返却作業を行う．free_bit を呼び出し，ゾーンマップとビット番号を引数として渡すだけである．free_bit は，空の i ノードを返却する際にも用いられるが，もちろんその場合は i ノードマップを最初の引数として用いる．

キャッシュの管理には，ブロックの読み書きが必要である．ディスクとの単純なインターフェイスのために，手続き rw_block(8295行目)が用意されている．この手続きは単一ブロックの読み書きを行うものである．同様に手続き rw_inode と rw_super が，それぞれ i ノードとスーパーブロックの読み書き用に用意されている．

このファイルの最後の手続きは invalidate(8326行目)である．これはディスクがマウント解除される時に使用され，マウント解除されるファイルシステムに属するすべてのブロックをキャッシュから除去する．この作業を行っておかないと，デバイスが再度利用された時(別のフロッピーディスクによって)，ファイルシステムは新しいディスクのブロックではなく，古いものを使ってしまう可能性がある．

■ i ノード管理

ブロックキャッシュ以外にも手続きの助けを必要とするテーブルがある．i ノードテーブルがそれである．多くの手続きは，ブロック管理の手続きと機能的に同じである．図5.37に一覧を示す．

get_inode	メモリ内のi-nodeを得る
put_inode	不必要なi-nodeを戻す
alloc_inode	新しいi-nodeを割り当てる(新しいファイル)
wipe_inode	i-node内のいくつかのフィールドのクリアする
free_inode	i-nodeを解放(ファイルの削除)する
rw_inode	メモリとディスク間のi-nodeを転送する
dup_inode	i-nodeが使用中であるかを示す

図5.37 i ノード管理用手続き

手続き get_inode(8379行目)は get_block に似ている．ファイルシステムの任意の部分が i ノードを必要とした場合，get_inode を呼び出し，それを得る．get_inode はまず i ノードテーブルを最初に検索し，i ノードの存在を確認する．存在が確認できたら，使用カウンタを増加してそれを指すポインタを返す．この検索作業は8389から8406行目に記述されている．i ノードがメモリ内に存在しない場合は，rw_inode を呼び出し，i ノードをロードする．

i ノードを必要とする手続きが終了した後，手続き put_inode(8421行目)を呼び出して i ノード

を戻す．これにより，使用カウンタ i_count を減少することになる．ここでカウントが 0 になった場合，ファイルが使用されなくなったことになり，i ノードはテーブルから削除される．内容が変更されている場合はディスクに書き込まれる．

i_link フィールドが 0 の場合は，ファイルを指しているディレクトリ・エントリが存在しないことになり，そのゾーンがすべて解放される．使用カウントが 0 になるのと，リンク数が 0 になるのとではその意味も，原因も，結果も全く異なる点に注意されたい．

新しいファイルが作成される場合，i ノードを割り当てなくてはならない．この作業は alloc_inode(8446 行目)によって行われる．ゾーンの場合はファイルごとに近くのゾーンを割り当てようとしたが，これと異なり任意の i ノードを使えばよい．

i ノードが得られたら，get_inode を呼び出して，i ノードをメモリ内のテーブルに取り出す．次にそのフィールドの一部が 8482 から 8486 行目で，また残りの部分は wipe_inode(8503 行目)で初期化される．ファイルシステムの他の部分で，特定の i ノードフィールド(全部ではない)をクリアするために wipe_inode が必要となるので，この様な作業分担が行われている．

ファイルを削除すると，free_inode(8525 行目)が呼び出され，そのi ノードが解放される．ここでは i ノードビットマップ内の該当するビットが 1 に設定されるだけである．

手続き rw_inode(8543 行目)は，rw_block と同じように，ディスクから i ノードを取り出す役割を担っている．以下の手順で作業を実行する．

①必要な i ノードを含むブロックを算出する．
②get_block を呼び出してブロックを読み取る．
③i ノードを抽出して，それを inode テーブルにコピーする．
④put_block を呼び出して，ブロックを返却する．

手続き dup_inode(8579 行目)は，i ノードの使用カウントを増加するだけである．

■ スーパーブロックの管理

ファイル super.c には，スーパーブロックとビットマップの管理を行う手続きが記述されている．このファイルには，図 5.38 で示している 7 つの手続きがある．

load_bit_maps	別のファイルシステムのビットマップを得る
unload_bit_maps	ファイルシステムのマウント解除後のビットマップを戻す
alloc_bit	ゾーンかi-nodeマップからのビットを割り当てる
free_bit	ゾーンおよびi-nodeマップ中のビットを解放する
get_super	デバイスのためのスーパーブロックテーブルをサーチする
scale_factor	ゾーンとブロックを変換するためにシフトするビット数
rw_super	メモリとディスク間のスーパーブロック転送

図 5.38 スーパーブロックとビットマップ管理用の手続き

load_bit_maps(8631 行目)はルートデバイスがロードされる時，または新しいファイルシステムがマウントされる時に呼び出される．すべてのビットマップ・ブロックを読み取り，スーパーブロックがそれらを指すように設定する．スーパーブロック内の配列 s_imap と s_zmap は，それぞれ i ノードビットマップ・ブロックとゾーン・ビットマップ・ブロックを指すものである．

ファイルシステムがマウント解除されると，そのビットマップが unload_bit_maps(8669 行目)によってディスクに書き込まれる．

i ノードまたはゾーンが必要な場合は，前述の様に alloc_inode または alloc_zone を呼び出す．これらは両方とも alloc_bit(8689 行目)を呼び出し，実際に関連するビットマップを検索する．検索には以下の三重のループが使用される．

- ビットマップの全ブロックに対する外部ループ
- ブロックの全ワードに対する中間ループ．
- ワードの全ビットに対する内部ループ

中間ループは現在のワードが 0 の補数，すべてのビットが 1 であるワードに等しいことを調べる．等しければ，空 i ノードもゾーンも存在しないことになるため，次のワードが調べられる．異なる値を持つワードが検出されたら，少なくても 1 つの 0 のビットが含まれていることになるので，内部ループに入り，空(すなわち 0)ビットを検出する．すべてのブロックを調べたが該当するものがなかった場合は，空の i ノードもゾーンも存在しないことになり，コード NO_BIT(0)が返される．

ビットの解放には検索が不要なため，割り当てより簡単である．free_bit(8747 行目)は，解放するビットを持つビットマップ・ブロックの算出を行い，そのビットを 0 に設定する．ブロック自体は常にメモリ内にあり，スーパーブロック内の s_imap または s_zmap ポインタによって指し示されている．

次の手続き get_super(8771 行目)は，特定のデバイスに対するスーパーブロックを検出する．例えばファイルシステムのマウントを行う場合，そのデバイスがまだマウントされていないことを確認しなければならない．この検査は get_super によりデバイスを検出することで実現する．デバイスが検出できなかった場合は，そのファイルシステムがまだマウントされていないことになる．

ブロックとゾーンの変換はブロック番号を左に，そしてゾーン番号を右にシフトして行う．シフトする量はゾーンあたりのブロック数に依存する．この数はファイルシステムによって異なる．手続き scale_factor(8810 行目)はこの値を求める．

最後に rw_super(8824 行目)は，前述した rw_block と rw_inode に似ており，スーパーブロックの読み書きを行うために呼び出される．

■ ファイル記述子の管理

MINIX ではファイル記述子と filp テーブルを管理するために，特別な手続きを設けている(図 5.35 参照)．これらはファイル filedes.c に記述されている．ファイルが作成，またはオープンされると，空のファイル記述子と空の filp スロット上が必要になる．手続き get_fd(8871 行目)を使ってこれらを検出することができる．CREAT や OPEN が成功すると確信できるまでには多くの検査が必要なため，使用中の印は付けられていないので注意されたい．

get_filp(8916 行目)はファイル記述子が指定された範囲内にあるかを確認し，その filp ポインタを返す．

このファイルの最後の手続きは find_filp(8930 行目)である．プロセスによる破損したパイプ(すなわち他のプロセスが読取り用にオープンしていないパイプ)への書込みを検知するために必要である．filp テーブルを検索して，読取りを行う可能性のあるものを検出する．

5.7.3 メインプログラム

ファイルシステムのメインループは，8992 行目から始まっているファイル main.c に記述されている．その構造は，メモリマネージャと入出力タスクのメインループに非常によく似ている．get_work を呼び出すと，次のメッセージの到着を待つことになる(パイプまたは端末上でそれまで保留状態にあったプロセスが処理される場合を除く)．大域変数 who を呼出し側のプロセステーブル・スロット番号に設定し，もう 1 つの大域変数 fs_call に，実行するシステムコール番号を設定する．

メインループに戻った時，3 つのフラグが設定される．fp は呼出し側のプロセステーブル・スロットを指し，super_user は呼出し側がスーパーユーザーか否かを示し，dont_reply は FALSE に初期化される．次に最も重要な作業が行われる．システムコールを実行する手続きを呼び出す．この手続きは fs_call を，手続きへのポインタ call_vector 配列に対するインデックスとして使用し，その手続きを呼び出す．

メインループに制御が戻った時，dont_reply が設定されていなければ返答は行われない(例えばブロックが空のパイプを読み出そうとしてブロックしてしまった)．それ以外の場合は，なんらかの返答が行われる．メインループの最後のステートメントは，連続的に読み取られるファイルを検出するために設けられたものであり，実際に要求がある前に次のブロックをキャッシュに読み込み，システムの性能を向上させる．

手続き get_work(9016 行目)は，これまでにブロックされた手続きのうち，再開できるものがないかを調べる．再開できる手続きがあれば，新しいメッセージより高い優先度を与えられる．特に内部で行う作業がない場合，ファイルシステムはメッセージを受け取るためにカーネルを呼び出す(9042 行目)．

システムコールが正常終了しても，または異常終了しても，reply(ライン 9053 行目)を用いて呼出し側に返答を行う．基本的には send が失敗することはないが，カーネルは確認のために状態

コードを返す．

ファイルシステムが実行を開始する前に，fs_init(9069 行目)によって自身の初期化を行う．この手続きによってブロックキャッシュが用いる連結されたリストが構築され，64K の境界を超えるようなバッファをすべて削除する(IBM PC の DMA チップは 64K の境界を超えることができないため)．そしてブートディスケットから RAM ディスクをロードし，スーパーブロック・テーブルを初期化し，ルートデバイスのスーパーブロックと i ノードを読み取る．すべてに問題がなければ，i ノードとゾーン・ビットマップがロードされる．最後にそれらの数値が正当であるかを確認する．

ブートディスケットを作成する時，RAM イメージのビット単位のコピーが MINIX バイナリの後に含まれる．手続き load_ram は，最初に各種設定(RAM ディスクドライバに RAM ディスクの配置先と，その大きさを伝えることも含まれている)を行い，ブロック単位で RAM ディスクにコピーする．

■ ディスパッチ・テーブル

ファイル table.c(9300 行目)は，どの手続きがどのシステムコール番号を処理するかを決定するために，メインループで使用されるポインタの配列を含んでいる．同様のテーブルがメモリマネージャ内にも存在していた．

ただし 9416 行目のテーブル dmap は，新たに設けられたものである．このテーブルは 0 から始まる 1 つの列を各種のデバイスに対して持っている．デバイスをオープン，クローズ，読み書きすると，操作を行う手続きの名前がこのテーブルによって提供される．これらの手続きはすべてファイルシステムのアドレス空間に位置している．これらのうち，大半は特に何も行わないが，なかには実際に入出力を行うタスクを呼び出すものもある．各種デバイスに該当するタスク番号も，このテーブルが提供する．

新しく主デバイスが MINIX に追加されると，このテーブルに 1 行が追加され，デバイスのオープン，クローズ，読み書きの際の処置について指示する．簡単な例として，テープドライブが MINIX に追加されたとしよう．その特殊ファイルがオープンされた時に，テーブル内の手続きによりテープがすでに使用中であるかどうかを調べることができる．

5.7.4 ファイルに対する操作

この項では，ファイル(ディレクトリではなく)に操作を行うシステムコールについて考察する．ファイルの作成，オープン，クローズからその読み書きまでを調べていこう．

■ ファイルの作成，オープン，クローズ

ファイル open.c は 5 つのシステムコールに対するコードを含んでいる．CREAT，MKNOD，OPEN，CLOSE，LSEEK がそれである．これらを 1 つずつ見ていくことにしよう．ファイルの

作成は以下の手順で行われる.

①新しいファイルを割当て，初期化する.
②対応するディレクトリに新しいファイルを追加する.
③新しいファイルのファイル記述子を設定し，返却する.

CREAT を処理する手続きは do_creat(9479 行目)である．メモリマネージャの場合と同様に，システムコール XXX は手続き do_xxx によって実行されるという規則をファイルシステム内でも用いている.

do_creat は，最初に新しいファイルの名前を取り出し，空のファイル記述子と filp テーブルスロットが使用可能であるかどうかを確認する．新しい i ノードを実際に作成するのは手続き new_node であり，これは 9496 行目で呼び出されている．i ノードを作成できなかった場合，new_node が大域変数 err_code を設定する.

do_creat が行う作業は，ファイルがすでに存在するか否かによって異なってくる．ファイルが存在しない場合，9504 から 9521 行目はスキップされ，テーブルスロットが要求されて，ファイル記述子が返される.

ファイルが存在する場合は，ファイルシステムはファイルの種類やモードなどを確認しなければならない．通常のファイルに対して CREAT を行うと，そのファイルは長さ 0 に切り捨てられる．書込み可能な特殊ファイルに対して CREAT を実行すると，書込みのためにオープンされることになる．それをディレクトリに対して実行すると，必ず拒否されてしまう.

do_creat のコードや，他の多くのファイルシステム手続きでは，各種のエラーや不正な状況を検出するためのコードがたくさん記述されている．これらのコードはエラーのない，強固なファイルシステムを作るには欠かせないものである．何も問題がなければファイル記述子と先頭に配置された filp スロットが，ここで割当て済みとして印付けされ，ファイル記述子が返される．最初から割当て済みとは印付けされず，必要に応じて途中で終了しやすいようにしている.

MKNOD システムコールは，do_mknod(9541 行目)によって処理される．この手続きは do_creat と似ているが，i ノードを作成し，そのためのディレクトリ・エントリを設けるだけである．i ノードがすでに存在する場合，システムコールはエラー終了する．do_creat で見たような，状況に応じた分析は必要ない.

i ノードの割当てとファイルシステムへのパス名設定は，new_node(9557 行目)によって行われる．9575 行目のステートメントは，パス名の解析(すなわちパス名の要素ごとに調べる)を最後のディレクトリまで行う．3 行後の advance に対する呼出しは，パス名の最後の要素をオープンできるかどうか調べるものである.

```
fd=creat("/user/ast/foober", 0755);
```

例えば上記の様な呼出しでは，last_dir は/user/ast の i ノードをテーブル内にロードし，それ

に対するポインタを返す．ファイルが存在しない場合，このi ノードを使ってディレクトリに foobar を追加することになる．ファイルの追加，削除を行う他のシステムコールも，すべて最初にパス内の最後のディレクトリをオープンするために last_dir を使用する.

new_node によってファイルが存在しないことが判明した場合，9581 行目で alloc_inode を呼び出し，新しい i ノードの割当てとロードを行い，それを指すポインタを返す．空の i ノードが残っていない場合，new_node は失敗し，NIL_INODE が返される.

i ノードが割り当てられれば，9591 行目で作業を継続し，フィールドのいくつかを埋め，ディスクに再度書き込み，ファイル名をパス名中の最後のディレクトリに書き込む(9596 行目)．ここでもファイルシステムは絶えずエラーを調べ，検出時には使用している i ノードやブロックなど，すべての資源を入念に解除する．例えば i ノードが不足した時，現在のコールの結果を取り消したり，呼出し側に対してエラーコードを返さずに，あえて MINIX をパニック状態にするつもりならば，ファイルシステムはかなり簡素化できるだろう.

次の手続きは do_open(9622 行目)である．各種検査を行った後，eat_path を呼び出し，ファイル名の解析を行い，i ノードをメモリ内にフェッチする．i ノードが使用可能になると，モードを調べてファイルがオープン可能かどうかを確認する．9645 行目における forbidden の呼出しでは，rwx ビットの検査を行う．ディレクトリと特殊ファイルは後で処理される．最後にファイル記述子が関数値として返される.

ファイルのクローズは，ファイルのオープンよりも簡単である．do_close によってこの作業が行われる(9680 行目)．パイプと特殊ファイルは若干注意しなければならないが，通常のファイルでは filp カウンタを減少し，0 になった場合は i ノードが put_inode を使用して戻される.

i ノードを戻すことにより，i ノードテーブル内のカウンタは減少される．したがって結果的にテーブルから削除されることになる．この操作は i ノードの解放(i ノードが未使用であることをビットマップに設定すること)とは違う．i ノードはファイルがそれを持つすべてのディレクトリから除去された場合に限り，解放することができる.

このファイルの最後の手続きは do_lseek(9721 行目)である．シークが実行されると，この手続きが呼び出され，ファイル位置を新しい値に設定する.

■ ファイルの読出し

ファイルをオープンすると，それを読み書きすることができる．最初に読出しについて説明し，次に書込みについて説明する．いくつかの点でこの 2 つの作業は異なっているが，do_read(9784 行目)も do_write(10125 行目)はどちらも共通の手続き read_write(9794 行目)を呼び出し，そこで大半の作業を行うという点では一致している.

9811 から 9818 行目までのコードはメモリマネージャが，ファイルシステムを使ってそのユーザー空間にセグメント全体をロードする際に用いる．通常のコールは 9821 行目から処理を開始され，そこでは正当性検査(例えば書込みだけが行えるファイルから読み取ろうとしたなど)と変数

の初期化が行われる．キャラクタ型特殊ファイルからの読出しはブロックキャッシュを通過しないため，9836行目で判別されている．

9844から9854行目までの検査は，デバイスの許容量よりも成長する可能性のあるファイルや，ファイル終端を超えて書込みを行うことによってファイル内に空間を作ってしまわないかなど，書込みに関するものである．これはMINIXの概要説明で触れたように，1つのゾーンに複数のブロックが存在すると，困難な問題を引き起こすことになるからである．パイプも特殊であり，検査が必要である．

少なくとも通常のファイルにおいては，読出し機構の主な処理となるのが9861行目から始まるループである．このループは要求をいくつかの処理単位(chunk)に分割し，それぞれが1つのディスクブロック内に収まるようにする．処理単位は，現在位置から始まり以下のいずれかの状況になるまでである．

- すべてのバイトを読み出した場合①
- ブロック境界に達した場合②
- ファイル終端に達した場合③

この様な規定は，1つの処理単位が2つのディスクブロックにまたがらないことを意味している．図5.39では，処理単位のサイズを決定する方法として3つの例を紹介している．実際の処理は9862から9871行目で行われている．

処理単位の実際の読出し作業はrw_chunk(9874行目)によって行われる．制御が戻された後，各種カウンタとポインタが増加され，次の操作を開始する．ループが終了した後，ファイル位置と他の変数を更新することができる(例えばパイプ・ポインタなど)．

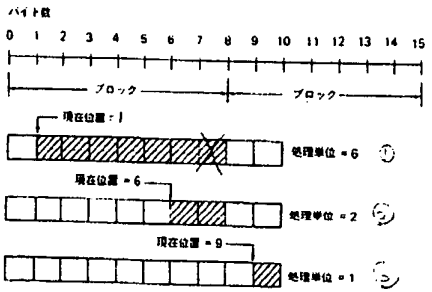

図5.39 10バイトのファイルに対して処理単位のサイズを決定する3つの例
ブロックサイズは8バイトであり，必要なバイト数は6である．処理単位は斜線で示してある．

先読みを指定した場合は，読み取るiノードと位置が大域変数に保存され，返答がユーザーに送られた後，ファイルシステムが次のブロックの処理を開始できるようにしておく．通常，ファイルシステムは次のディスクブロックを待ってブロックし，その間にユーザープロセスはすでに与えられているデータの処理を行うことができる．

手続きrw_chunk(9919行目)は，iノードとファイル位置を指定し，それらを物理ディスクブロック番号に変換し，そのブロック(またはそのブロックの一部)をユーザー空間に移動するよう指示する．相対的なファイル位置と物理ディスクアドレスの対応付けは，iノードと間接ブロックを関知しているread_mapが行う．通常のファイルの場合は，9944と9945行目の変数bおよびdevに物理ブロック番号と，デバイス番号がそれぞれ格納されることになる．9964行目で呼び出しているget_blockは，キャッシュハンドラに対してブロックの検出を依頼するものであり，必要に応じてそれを読み取る．

ブロックに対するポインタを得ると，9972行目のrw_userの呼出しにより，要求した部分をユーザー空間に転送する．そしてブロックをput_blockによって解除し，後で必要に応じてキャッシュから削除できるようにする(get_blockで要求すると，ブロックのヘッダ内にあるカウンタはそれが使用中であることを示し，削除の対象からはずす．そしてput_blockはカウンタ値を減少させる)．

read_map(9984行目)はiノードを調べることによって，論理ファイル位置を物理ブロック番号に変換する．ファイルの先頭から最初の7ゾーン(iノード内に含まれている)のうちいずれかに存在するブロックに関しては，どのゾーンが必要であり，そしてそれがどのブロックであるかの決定は，簡単な処理で行える．それ以上のブロックを使用しているファイルに関しては，1つ以上の間接ブロックを読み取らなければならない．

手続きrw_user(10042行目)は，単にシステムタスク用にメッセージを作成し，送信するものである．実際のコピー作業は，カーネルが行う．ファイルシステムがコピーを行うことはない．ファイルシステムはユーザーがメモリ内のどこに位置しているのかすら知らない．この様な余分なオーバーヘッドは高度にモジュール化されたシステムで，払わなくてはならない犠牲である．

最後にread_ahead(10082行目)は論理位置を物理ブロック番号に変更し，get_blockを呼び出してブロックがキャッシュ内に存在することを確かめ，ブロックを即座に戻す．結局，ブロックに対して何も行わないのである．ブロックをすぐに利用する時のために，ブロックをできるだけキャッシュ内に置こうというものである．

read_aheadは，main内のメインループでしか呼び出せない点に注意されたい．READシステムコールの処理の一部として呼び出すことはできない．read_aheadの呼出しは必ず返答を終えてから行い，先読み中にファイルシステムがディスクを待機しなければならない場合でも，ユーザーが実行を継続できるようにする．図5.40はファイルの読出しに関する主な手続きの関係を図示したものである．

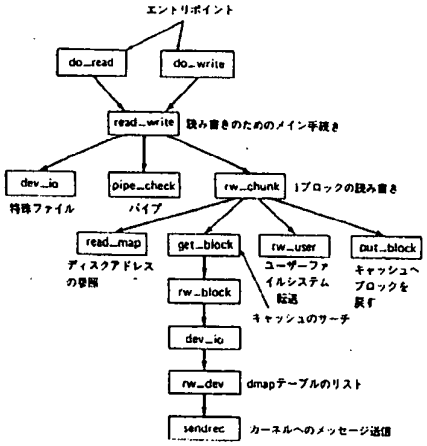

図5.40 ファイルの読出しに関するいくつかの手続き

■ ファイルの書込み

ファイルの書込みは，読取りと似ている．ただし書込み作業では，新しいディスクブロックを割り当てなければならない．write_map(10135行目)はread_mapと似ているが，iノードと間接ブロック内の物理ブロック番号を検索する代わりに，新しいブロック番号をiノードに格納する(正確にいえば，ブロック番号ではなく，ゾーン番号を格納する)．

write_mapのコードはいくつかの状況に対応するため，若干長めで，細かくなっている．ゾーンがファイルの先頭に近ければ，iノードに格納される(10160行目)．

一番面倒なのは，ファイルが単一間接ブロックで扱うことのできる最大の大きさとなった場合である．この場合二重間接ブロックが割り当てられる．次に単一間接ブロックの割当てが必要であり，そのアドレスが二重間接ブロックに置かれる時に，二重間接ブロックの割当ては成功したが，単一間接ブロックを割り当てることができない場合(すなわちディスクが満杯の場合)，ビットマップを破壊しないように，注意深く二重間接ブロックを解放しなければならない．

ここでも，パニック状態となって構わないのであれば，コードはかなり簡潔化できる．しかしユーザーの立場からいえば，ディスク空間が足りなくなっても，コンピュータのクラッシュや，ファイルシステムの破壊を発生せずに，WRITEからのエラーを返されたほうがどれだけ助かるかわからない．

write.c内の次の手続きはclear_zoneである．これはファイル終端を超えてシークを行った場合に，データを書き込む前に実行され，ブロックの消去を処理する手続きである．幸運にも，こ

の様な状況はそれほど頻繁には発生しない．

new_block(10265行目)は，新しいブロックが必要になった場合には必ずrw_chunk内に記述されている9955行目で呼び出される．図5.41はシーケンシャル・ファイルの成長を5段階に分けて示したものである．ブロックサイズは1Kで，ゾーンサイズは2Kとなっている．

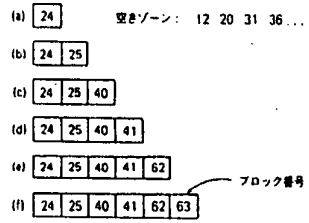

図5.41 シーケンシャル・ファイルの成長
(a)－(f) 1Kブロックの連続割当て．ゾーンは2Kである．

new_blockが初めに呼び出された時は，ゾーン12(ブロック24と25)を割り当てる．次に呼び出された時は，すでに割り当てられているがまだ使用されていないブロック25を使用する．3回目の呼出しでは，ゾーン20(ブロック40と41)が割り当てられ，といった具合になる．zero_block(10318行目)はブロックをクリアし，それまでの内容を削除する．

■ パイプ

パイプも通常のファイルと多くの共通点を持つ．しかしこの項では相違点に焦点を当てることにする．まずパイプはCREATではなく，PIPEシステムコールによって作成される．PIPEコールはファイルpipe.c内のdo_pipe(10384行目)によって処理される．do_pipeの実際の作業はパイプにiノードを割り当て，そのファイル記述子を2つ返すだけである．

パイプの読み書きも，ファイルの読み書きとは若干異なっている．パイプの容量は有限だからである．すでに満杯のパイプに書込みを行おうとすると，保留状態になってしまう．同様に空のパイプからの読出しも，保留状態を引き起こす．そのためパイプには，現在位置(読み出そうとするものが使用)と，サイズ(書き込もうとするものが使用)の2つのポインタが存在し，データをどこから書き込み，どこから読み出すかを決定できるようになっている．

パイプに対する操作が可能かどうかを調べる各種検査がpipe_check(10433行目)によって提供されている．呼出し側を保留状態にする可能性のあるこれらの検査以外にも，pipe_checkを用いてreleaseを呼び出し，データ型またはデータ不足によって保留になっていたプロセスが再開できるか調べることもできる．10457行目と10458行目では，それぞれスリープ中の読出しおよび

滞込みプロセスを再開をしている．破損したパイプ(読み出す者のいないパイプ)もここで検出することができる．

プロセスの保留はsuspend(10488行目)によって行われる．引数をプロセステーブル内に保存し，フラグdont_replyをTRUEに設定して，ファイルシステムの返答メッセージを禁止する．

手続きrelease(10510行目)は，パイプ上でそれまで保留になっていたプロセスが，実行を継続できるかどうかを調べるために呼び出される．実行を継続できるようなプロセスが存在する場合，reviveを呼び出した後でメインループがそれを知れるように，フラグを設定する．

pipeの最後の手続きはdo_unpause(10572行目)である．メモリマネージャがプロセスにシグナルを送ろうとする時，そのプロセスがパイプや，特殊ファイルが原因でスリープしていないかを確認する(その様な場合EINTRエラーを用いて，目覚めさせなくてはならない)．メモリマネージャはパイプまたは特殊ファイルに関して何も知らないため，ファイルシステムに質問のメッセージを送信する．そのメッセージはdo_unpauseによって処理され，ブロックしていたプロセスがあればそれを再開する．

5.7.5 ディレクトリとパス

ファイルがどの様にして読み書きされているかを勉強したところで，次にパス名とディレクトリの処理について見てみよう．

■ パスからiノードへの変換

多くのシステムコール(例えばOPEN, UNLINK, MOUNTなど)は引数としてパス名(すなわちファイル名)を持っている．これらの多くは本来の作業に取りかかる前に，指定されたファイルのiノードを得なければならない．ここではパス名がどの様にしてiノードに変換されているのかを詳しく見ていく．概要については図5.11ですでに学んでいる．

パス名の解析は，ファイルpath.cで行われる．最初の手続きeat_path(10675行目)はパス名を指すポインタを受け取り，それを解析し，iノードを指すポインタを返す．last_dirを呼び出してパス名中の最後のディレクトリのiノードを取り出し，次にadvanceを呼び出してパス名の最後の要素を得る．検索が例えばパス名内の1つのディレクトリが存在しないため，あるいは存在はするが検索できないような保護モードが指定されていたために失敗した場合，NIL_INODEがiノードを指すポインタの代わりに返される．

パス名は絶対パス名でも，相対パス名でも構わない．またスラッシュで区切られた任意の数の要素を持つことができる．これらの作業はlast_dir(10703行目)によって処理される．パス名の最初の文字を調べ，相対パス名か絶対パス名かを識別することから始まる(10703行目)．そしてripに，絶対パス名に対してはルートiノードが設定され，相対パス名に対してはカレント・ワーキングディレクトリのiノードが設定される．

この時点でlast_dirは，パス名と共に，最初の要素を検索するディレクトリのiノードを指すポインタを得る．10726行目でループに入り，パス名を要素ごとに解析していく．最後に到達すると，最後のディレクトリを指すポインタを返す．

get_name(10749行目)は文字列から要素を抽出するための手続きである．より興味深いのはadvance(10792行目)で，引数としてディレクトリ・ポインタと文字列が与えられ，ディレクトリからその文字列を検出する．文字列を検出すると，advanceはそのiノードを指すポインタを返す．マウントされたファイルシステムに関する処理がここで行われる．

advanceは文字列を検索するが，実際に文字列とディレクトリ・エントリとの比較を行うのはsearch_dirである．search_dirはファイルシステム内で唯一ディレクトリ・ファイルの検査を行う場所である．入れ子になった2つのループがそこに含まれており，そのうち1つはディレクトリ内のブロックに対して，もう1つはブロック内のエントリに対してループを行うものである．手続きsearch_dirは，ディレクトリに名前を追加したり，削除したりする場合にも使われる．図5.42はパス名の検索に使用される主な手続きの関係を示したものである．

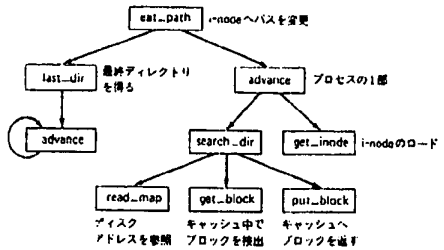

図5.42 パス名検索用のいくつかの手続き

■ ファイルシステムのマウント

ファイルシステム全体に影響を及ぼすシステムコールとして，MOUNTとUMOUNTの2つが提供されている．これらは異なる副デバイス上にある個々のファイルシステムを，"接続して"単一のツリーを形成するものである．図5.31でも見たように，マウントはマウントするファイルシステムのルートiノードと，スーパーブロックを読み取り，そのスーパーブロックに2つのポインタを設定することによって実行される．そのうちいずれかはマウントされる側のiノードを指し，他方はマウントするファイルシステムのルートiノードを指す．

これらのポインタ設定は，ファイルmount.cの11116と11117行目において，do_mountを使って行われる．ポインタの設定前に記述されている2ページ分のコードは，大半がファイルシステムのマウント中に発生するエラーの検出を行うためのものである．そのうち主なものを以下にあげる．

●指定した特殊ファイルはブロック型デバイスでない．
●特殊ファイルはブロック型デバイスであるが，すでにマウントされている．
●マウントしようとしているファイルシステムに不正なマジックナンバーが与えられている．
●マウントしようとしているファイルシステムは無効である(例えばiノードがない)．
●マウントされる側のファイルが存在しない，または特殊ファイルである．
●マウントされたファイルシステムにビットマップのための空間がない．
●マウントされたファイルシステムにスーパーブロックのための空間がない．
●マウントされたファイルシステムにルートiノードのための空間がない．

おそらくこの様な点に固執することはあまり適当ではないように思えるだろう．しかし実践的なオペレーティング・システムでは必ずといっていいほど，かなりの量のコードをそれほど面白くも，また重要とも思われないような作業に費やしているのが現実である．しかしこの様な作業こそシステムの質を向上させるのに欠かせないものなのである．ユーザーが1箇月に一度の割合で間違ったフロッピーディスクを誤ってマウントしようとし，これがクラッシュやファイルシステムの破壊を招くとしたら，ユーザーはこのシステムの信頼性に疑問を感じ，自分自身ではなくシステム設計者を非難するだろう．

トーマス・エジソンはこれに関連した言葉を残している．彼は「天才」が1%のひらめきと，99%の努力で作られるものだと述べている．良いシステムと，悪いシステムの違いはスケジューリング・アルゴリズムの素晴らしさではなく，詳細部分への心配りにあるのである．

ファイルのマウント解除は，ファイルのマウントより簡単である．間違いやすい部分が少ないからである．重要な点は，削除されるファイルシステム上に，オープンしたままのファイルやワーキングディレクトリを持つプロセスが存在しないかを確認することである．この検査はいたって簡単である．iノードテーブル全体を走査し，メモリ内のiノードのうち削除しようとするファイルシステムに属しているものがないか調べるだけでよい(ルートiノードを除く)．その様なiノードが存在すれば，UMOUNTシステムコールは失敗に終わる．

mount.cの最後の手続きは，name_to_dev(11180行目)である．この手続きは特殊ファイルを指定し，そのiノードを得て，その主デバイスおよび副デバイス番号を取り出す．これらはiノード内にあり，通常最初のゾーンが向かう場所に保存されている．このスロットは特殊ファイルがゾーンを持たないため，提供されている．

■ ファイルのリンクとリンク解除

次のファイルはファイルのリンクとリンク解除を行うlink.cである．手続きdo_link(11275行目)は，大半のコードがエラーの検査用に費やされているという点でdo_mountに似ている．

```
link(file_name, link_name);
```


上記の呼出しにおいて発生する可能性があるエラーの主なものを以下に挙げる．

- file_nameが存在しない，またはアクセスできない．
- file_nameがすでに最大数のリンクを行っている．
- file_nameがディレクトリである(スーパーユーザーのみがリンクできる)．
- file_nameがすでに存在する．
- file_nameとlink_nameが異なるデバイスである．

エラーが存在しなければ，新しいディレクトリ・エントリが文字列link_nameと，file_nameのiノード番号を使って作成される．実際の入力は，searchdirによって11324行目のdo_linkから行われる．

ファイルはリンク解除することによって削除される．作業はdo_unlink(11342行目)によって行われる．ここでも各種の検査が最初に行われる．エラーがなければ，ディレクトリ・エントリがクリアされ，iノード内のリンク数が1つ減少される．

リンクカウントがここで0になった場合，すべてのゾーンがtruncate(11388行目)によって解放される．この手続きはiノードから検出したゾーンそれぞれを解放するという，簡単な方法で処理されている．

5.7.6 その他のシステムコール

最後に，状態，ディレクトリ，保護，時間，その他各種の機能に関わる雑多なシステムコールを見ていくことにする．

■ ディレクトリとファイル状態の変更

ファイルstadir.cにはCHDIR，CHROOT，STAT，FSTATという4つのシステムコールに関する処理が記述されている．10722行目のlast_dirでは，パスの最初の文字が，スラッシュであるかを調べることによってパス検索が始まることを学んだ．結果によってポインタがワーキングディレクトリか，ルートディレクトリに設定される．

1つのワーキング(またはルート)ディレクトリから別なディレクトリに移ることは，呼出し側のプロセステーブルのこの2つのポインタを変更することである．この変更はdo_chdir(11475行目)とdo_chroot(11500行目)によって行われる．どちらも必要な検査を行ってから，change(11515行目)を呼び出し，古いディレクトリに置き換えられる新しいディレクトリをオープンする．

11483から11490行目のコードは，ユーザープロセスが行うCHDIRでは実行されない．これはメモリマネージャ用に特別に設けられているもので，EXEC処理を目的としてユーザーディレクトリに移ることができる．ユーザーが例えばa.outなどのファイルを，自身のワーキングディレクトリ内で実行しようとすると，メモリマネージャにとってはその場所を調べるよりも，そのディ

レクトリに移動したほうが簡単である.

このファイル内で扱われている残り2つのシステムコールSTATとFSTATは, ファイルの指定方法を除けば, 基本的には同じである. 前者はパス名を使って指定し, 後者はオープンされたファイルのファイル記述子を使用する. トップレベルの手続きである do_stat と do_fstat は, どちらも stat_inode を呼び出して作業を行う. stat_inode を呼び出す前に, do_stat はファイルをオープンすることで i ノードを指定する. この様にして do_stat と do_fstat は, stat_inode に i ノードポインタを渡す.

stat_inode の唯一の役割は i ノードから情報を取り出し, それをバッファ内にコピーすることである. バッファは1つのメッセージ内に収めきれないため, rw_user を用い, 11624行目でコピーしなければならない.

■保 護

MINIX の保護機構では, rwx ビットを使用している. この一連のビットは各ファイルに対して与えられており, それぞれ所有者と, そのグループ, そして一般ユーザーごとに分れている. ビットの設定は CHMOD システムコールによって行う. このシステムコールは do_chmod (11677行目)が実行する. 一連の妥当性検査を行った後, モードが 11704 行目で変更されている.

システムコール CHOWN は, 特定のファイルの内部 i ノード領域を変更するという点で, CHMOD に似ている. do_chown (11715行目)はスーパーユーザーしか実行できないが, 2つの実現方法も類似している.

システムコール UMASK を使って, ユーザーはマスク(プロセステーブル内に保存されている)を設定し, 以降の CREAT システムコールにおいて保護ビットのマスクを行う. これは 11752 行目のステートメントのみで行われる. ただしコールは古いマスク値を結果として返さなければならない. この様な余分な作業が, 必要なコード数を3倍に増やしてしまう(11751 から 11753 行目).

システムコール ACCESS を用いて, プロセスは指定した方法で(例えば読取り用に)ファイルのアクセスを行えるかどうかを確認できる. do_access (11760行目)を使って, ファイルの i ノードを得, 内部手続き forbidden (11782行目)を呼び出して, アクセスが禁止されていないかを調べる. forbidden は, uid と gid のみではなく, i ノード内の情報も調べる. その内容によって rwx グループのいずれかを選択し, アクセスの許可状態を調べる.

11782行目の read_only は, 引数として渡される i ノードが存在するファイルシステムが, 読取り専用または読み書き両用のどちらでマウントされているかを調べるものである. これは, 読取り専用にマウントされたファイルシステムに対する書込みを防ぐために必要である.

■時 刻

MINIX には時刻に関するシステムコールがいくつか提供されている. UTIME, TIME, STIME, TIMES である. これらを図5.43にまとめておく. ファイルシステムが処理しなければならない理由は特になく, たまたまそうなっただけである.

UTIME	ファイルの最終更新時刻をセット
TIME	現在の時刻を秒で得る
STIME	時刻の設定
TIMES	プロセスのアカウントを得る

図5.43 時間に関する4つのシステムコール

各ファイルに持たせているのは, ファイルが最後に修正された時刻を記録する32ビットの数値である. この時刻は i ノードに保存されている. ファイルの所有者, またはスーパーユーザーは, システムコール UTIME を使いこれを変更できる. 手続き do_utime (11877行目)は, i ノードを得, ユーザーが指定した時間をその i ノードに格納することによって, このシステムコールを実行している.

ファイルシステムでは時刻の保存を行っていない. カーネル内のクロックタスクがそれを保存している. その結果, 時刻を調べたり, 設定するためにはクロックタスクにメッセージを送るしか方法はない. 事実 do_time と do_stime はどちらもそのために提供されているものである. 時刻は 1970年1月1日からの時間(エポックと呼ばれる)が, 秒単位で保存されている.

アカウント情報もカーネル内に保存されている. 各クロック刻みごとに, 刻み分の課金(acount)をあるプログラムに対して行う. この情報はシステムタスクにメッセージを送ることで設定できる. do_tims(11937行目)がこの作業を行っている. たいていの C コンパイラは, 外部シンボルの先頭に下線を付け, またたいていのリンカではシンボルを8文字に切り捨てるため, do_time と do_times が識別できなくなってしまう. したがって, この手続きには, do_times という名前を付けていない.

■ その他

ファイル misc.c には, 他に適切な場所がないシステムコールの手続きをいくつか記述している. システムコール DUP は, ファイル記述子を複写するものである. すなわちその引数と同じファイルを指す新しいファイル記述子を作成する.

DUP とは呼出し形式が若干異なったシステムコールもある. 次の形式で呼出しを行う.

```
dup2(fd, fd2);
```

ここでfdはオープン済みのファイルを示すファイル記述子であり，fd2はファイルの割当てが行われていない整数である．この呼出しは，fd2をfdと同じファイルに対する有効ファイル記述子にするものである．

2つのシステムコールは同じメッセージ型を共有する．この2つは，DUP2のためにfdに設定された0100ビットによって識別される．どちらのコールもdo_dup(11981行目)によって処理される．作業は簡単でファイル記述子とfilpエントリを操作するだけである．

システムコールSYNCは，ディスクにロードされてから修正されたすべてのブロック，iノード，スーパーブロックをコピーするものである．処理はdo_sync(12018行目)によって行われる．該当するエントリを検出するために，すべてのテーブルを検索する．

システムコールFORK，EXIT，SETは，実際にはメモリマネージャのシステムコールであるが，結果をここで提示しなくてはならない．プロセスがフォークを行うと，カーネル，メモリマネージャ，ファイルシステムに通知しなくてはならない．これらの"システムコール"は，ユーザープロセスからではなく，メモリマネージャから発行される．関連する情報を記録するのがその役目である．

システムコールではないが，misc.cの最後のdo_revive(12419行目)は，ファイルシステムが要求された作業(例えばユーザープロセスへの入力データ提供など)を行うことのできなかったシステムタスクが作業を完了した時に呼び出される．ファイルシステムはここでプロセスを復元し，返答メッセージを送る．

5.7.7 入出力デバイスインターフェイス

MINIXにおける入出力操作は，カーネル内のタスクにメッセージを送信することによって行われている．これらのタスクに対するファイルシステムのインターフェイスは，ファイルdevice.c内に含まれている．このファイルには特殊ファイルに特別な操作を行う手続きも含まれている．最初に特殊ファイルをオープンすると，特別な操作を必要とする場合に備え，手続きdev_open(12233行目)が呼び出される．この手続きは特殊ファイルの主／副デバイス番号を取り出し，主デバイス番号をtable.cファイルのdmapテーブルに対するインデックスとして，ファイルシステム内の特別な処理を行うための手続きを呼び出す(12240行目)．通常，ここには特に何の作業も行わないno_callを設定するが，必要に応じてdmapに他の手続きを設定することもできる．

デバイスのクローズも同様である．この場合，dev_close(12261行目)が作業を行う．

実際にデバイスの入出力操作を必要とする場合は，dev_io(12261行目)が呼び出される．そして標準メッセージ(3章の図3.15参照)を作成し，それを指定したタスクに送る．read_writeがキャラクタ型特殊ファイルの処理用に呼び出され，rw_blockがブロック型特殊ファイル用に呼び出される．dev_ioがタスクからの回答を待っている間，ファイルシステムも待機する．内部にマルチプログラミング機能がないからである．ただしこの様な待機時間は通常非常に短いものである(最悪の場合でも2，300ミリ秒)．

device.cでは，IOCTLと呼ばれるシステムコールだけを処理している．このシステムコールはタスク・インターフェイスと密接な関係にあるため，ここに配置されている．IOCTLを実行すると，do_ioctlが呼び出され，メッセージを作成した後，適切なタスクに送信する．

find_dev(12328行目)は，主／副デバイス番号をデバイス番号から抽出するためのものである．device.cの最後の3つの手続きは，ファイルシステム内では特に呼び出されることはない．3つともdmapから，間接的に呼び出される．読み書きにはrw_devまたはrw_dev2を使用する(コールの階層については図5.41を参照されたい)．何も行わない手続きが必要な場合は，no_callが呼び出される．

5.7.8 汎用ユーティリティ

ファイルシステムには各種の場所で使用されるいくつかの汎用ユーティリティが存在する．これらはファイルutility.c内にまとめられている．最初の手続きはclock_timeである．メッセージをクロックタスクに送り，現在の時刻を調べるものである．次の手続きはcmp_stringで，2つの文字列を比較する．そして次のcopyは，ファイルシステムのアドレス空間の一部のデータブロックを，別な場所に移動する．

手続きfetch_nameは，大半のシステムコールが引数としてファイル名を持っているために必要となる．①ファイル名が短ければ，ユーザーからファイルシステムに対して送られるメッセージに含ませることができる．ファイル名が長ければ，ユーザー空間内にあるファイル名を指すポインタがメッセージに含まれる．fetch_nameはどちらの場合も検査を行い，結果として名前を得る．

no_sysはエラーハンドラであり，ファイルシステムに存在しないようなシステムコールを受け取った場合に呼び出される．最後にpanicは，メッセージを出力し，何か重大な問題が発生したことをカーネルに伝えるためのものである．

最後のファイルはputc.cである．2つの手続きがここに含まれているが，どちらもメッセージの印字に関わっている．標準のライブラリ関数は，ファイルシステムにメッセージを送信するため，使用することはできない．これら手続きはメッセージを直接端末タスクに送信する．

5.8 まとめ

ファイルシステムをその外側から眺めてみると，ファイルとディレクトリの集合，そしてそれを扱うための操作から構成されているように見える．ファイルは読取り，書込みが行え，ディレクトリは作成，削除が行える．そしてファイルを1つのディレクトリから別のディレクトリに移動することもできる．現在，多くのファイルシステムでは，階層型のディレクトリ・システムをサポートしており，ディレクトリが際限なくサブディレクトリを持つことができる．

内側から見たファイルシステムは，また違った顔を持っている．ファイルシステム設計者は記憶領域の割当て方法，ファイルとそれに含まれるブロックの管理などにも配慮しなければならない．またシステムによって各種のディレクトリ構造が存在することも学んだ．ファイルシステムの信頼性と性能も重要な問題であることがわかった．

最近のネットワーク・ファイルサーバー構築法についても，そのいくつかを学んだ．アトミック更新とトランザクションは，システムで最も重要な機能であることが多い．

セキュリティと保護は，システムユーザーにとっても，また設計者にとっても非常に重要な問題である．古くからあるシステムに見られるセキュリティの問題点と，システム全般に見られる一般的な問題点について解説した．またパスワード，アクセス制御リスト，その他の機能を用いた認証作業についても触れた．

最後にMINIXファイルシステムの詳細について学んだ．そしてその規模は大きいが，決して複雑ではないことを知った．ユーザープロセスからの作業要求を受け入れ，手続きへのポインタのテーブルを使って，要求されたシステムコールを実行するための手続きを呼び出す．そのモジュール化された構造と，カーネルの外という位置付けによって，MINIXから切り離して若干の修正を加えれば，独立したネットワーク・ファイルサーバーとして用いることもできる．

練習問題

1. あるオペレーティング・システムでは，システム全体で1つのディレクトリしかサポートしていないが，そのディレクトリが任意の長さのファイル名と，任意の数のファイルを持つことはできる．階層型のファイルシステムと呼べるようなものをシミュレートすることができるか．できるならば，その方法を示せ．

2. ファイル/etc/passwdに対し，5つの異なるパス名を挙げてみよ(ヒント：ディレクトリ・エントリ"."と".."を使用する)．

3. 空ディスク領域は空リストまたはビットマップを使って管理することができる．ディスクアドレスはDビットを必要としている．Bブロックから成るディスクのうち，Fブロックは空状態である．ここで空リストがビットマップより少ない空間を使用するような状態をあげてみよ．16ビット値をDに割り当てた場合，ディスクの利用率を計算せよ．

4. MS-DOSコンピュータはFATを使ってディスクブロックの記録をとっている．その性能を，長いファイルに対するランダムなシークに関してUNIXと比較せよ．以下の2つの場合について答えよ．
   (a) FATが必ずディスク上に存在する．
   (b) FATが必ずメモリ内に存在する．

5. UNIXでパスgames/zapperをオープンするのに必要なディスク参照回数は？

6. UNIXのiノードにおけるリンク数は冗長機能である．iノードを指しているディレクトリ・エントリの数を知らせるのが唯一の役割であるが，ディレクトリを調べれば同じことがわかる．それではなぜこの様な機能が設けられているのか．

7. あるUNIXファイルシステムでは，1,024バイトブロックと，16ビットのディスクアドレスが用いられている．iノードはデータブロック用の8つのディスクアドレスと，1つの単一間接ブロックアドレス，そして1つの二重間接ブロックアドレスを格納できる．最大ファイルサイズを良く考えて求めよ．

8. UNIXでは2つのディレクトリが同じiノードを指すことにより，ファイルの共有が行われている．MS-DOSではiノードが存在しない．MS-DOSでは同時に2つのディレクトリにファイルをリンクすることができるか．できるならばその方法を説明せよ．